docgrabnrun Documentation

Release 1.0

Luca Falsina

May 24, 2016

1 Quick start and tutorial **3**
1.1 Quick Setup 3
1.2 Tutorial 4

2 Why should I use Grab’n Run? **11**

3 Discussion of an example project **13**
3.1 Retrieve the example code and the emulator 13
3.2 List of example containers 22
3.3 MainActivity.java 22
3.4 DexClassLoader (apk) vs SecureDexClassLoader (apk) 22
3.5 DexClassLoader (jar) vs SecureDexClassLoader (jar) 24

4 Complementary topics **25**
4.1 Handle containers whose classes come from different package names which have a common relevant prefix 25
4.2 Handle containers whose classes come from different package names with no relevant common prefix 27
4.3 Reverse package name to obtain remote certificate URL 28
4.4 Perform dynamic code loading concurrently 28
4.5 On library developer side: how to prepare a valid library container compatible with GNR 31
4.6 Let GNR automatically handle library updates silently 37

5 Repackaging tool **41**
5.1 Use 41
5.2 Configuration 41

6 Indices and tables **47**

Grab'n Run (aka **GNR**) is a **simple** and **effective** Java Library that you can add to your Android projects to secure *dynamic class loading* operations.

For a **quick start** on how to include the library and how to use it in your projects give a look at Quick start and tutorial.

For a brief explanation on the **issue** of *insecure dynamic class loading* and on Grab'n Run purpose check the Why should I use Grab'n Run? section.

A concise **example** of use of the library is provided into an Android toy-application here. A *full explanation* of key extracts of this code is given into the Discussion of an example project section.

For a description on Grab'n Run **API** in *JavaDoc* style please refer to the API documentation.

For those willing for more **technicalities** and **advanced features** implemented in *Grab'n Run*, the section on Complementary topics is a *must-read*. This part of the documentation can also be used for **reference** as it presents how to handle properly some **tricky situations** that may occur while using *GNR*.

For an introduction on how to use the POC script for rewriting your application automatically to use the secure Grab'n Run API instead of the regualr ones for dynamic code loading, check out the Repackaging tool section.

CHAPTER 1

Quick start and tutorial

In this section you will see how to *retrieve and include Grab'n Run library* into your project (either by using Android Studio ot the Android Development Tool). After this setup step a **brief tutorial** will explain how to use classes in the library to **secure** the *dynamic code loading operations*.

Since this section is **introductory** and more descriptive, it should be read by those who are not familiar with this library or more in general with *class loading* in Android. On the other hand the Complementary topics section provides a more complete and detailed view on *Grab'n Run* library and its insights, while Discussion of an example project shows a simple use case of the concepts introduced here.

1.1 Quick Setup

Setting up GNR as an **additional library** for your *Android application* is very easy:

1.1.1 Android Studio (AS)

1. Modify the *build.gradle* file in the *app* module of your Android project by adding the following *compile* line in the *dependencies* body:

```
dependencies {
        // Grab'n Run will be imported from JCenter.
        // Verify that the string "jcenter()" is included in your repositories block!
        compile 'it.necst.grabnrun:grabnrun:1.0.4'
}
```

2. Resync your project to apply changes.

1.1.2 Android Development Tool (ADT)

1. Download the latest version of the *JAR* container of Grab'n Run.
2. Include the *JAR* in the **libs** subfolder of your Android project.

1.1.3 Adding missing permissions (Both AS and ADT)

Finally it is required to modify the *Android Manifest* of your application by adding a couple of required permissions if they are not already in place:

```
<manifest>
        <!--    Include following permission to be able to download remote resources
                like containers and certificates -->
        <uses-permission android:name="android.permission.ACCESS_NETWORK_STATE" />
        <!--    Include following permission to be able to download remote resources
                like containers and certificates -->
        <uses-permission android:name="android.permission.INTERNET" />
        <!--    Include following permission to be able to import local containers
                on SD card -->
        <uses-permission android:name="android.permission.READ_EXTERNAL_STORAGE" />
        ...
</manifest>
```

1.2 Tutorial

This tutorial assumes that you have **already retrieved Grab'n Run and linked it** to one of your existing Android projects.

1.2.1 Using standard DexClassLoader to load code dynamically

Let us pretend that you want to dynamically load an external class through DexClassLoader, a class in the *Android API* used to load classes from *jar* and *apk* files containing a **classes.dex** entry. This is a convenient way to execute code not installed as part of an application package.

Let's assume, for example, that you want to load an instance of `com.example.MyClass` located in the container *exampleJar.jar*, stored in the *Download* folder of the sd_card on the target phone. Note that this scenario may potentially lead to a **code injection** attack when you use the standard `DexClassLoader` since you are choosing to load code from a container which is stored in a **world writable location** of your phone. Notice that this kind of attack would be prevented with `SecureDexClassLoader`. Anyway a snippet of code to achieve this task is the following:

```
MyClass myClassInstance = null;
String jarContainerPath =       Environment.getExternalStorageDirectory().getAbsolutePath()
                                + "/Download/exampleJar.jar";
File dexOutputDir = getDir("dex", MODE_PRIVATE);
DexClassLoader mDexClassLoader = new DexClassLoader(    jarContainerPath,
                                                        dexOutputDir.getAbsolutePath(),
                                                        null,
                                                        getClass().getClassLoader());

try {
        Class<?> loadedClass = mDexClassLoader.loadClass("com.example.MyClass");
        myClassInstance = (MyClass) loadedClass.newInstance();

        // Do something with the loaded object myClassInstance
        // i.e. myClassInstance.doSomething();

} catch (ClassNotFoundException e) {
        e.printStackTrace();
} catch (InstantiationException e) {
        e.printStackTrace();
} catch (IllegalAccessException e) {
        e.printStackTrace();
}
```

The String `jarContainerPath` contains the path to *examplejar*, while `dexOutputDir` is an **application-private**, writable directory to cache optimized *dex* classes into *examplejar*. As reported in `DexClassLoader` documentation, you can retrieve `dexOutputDir` in different ways but it is fundamental that this cache folder is application-private; otherwise your application may be subjected to **code injection attacks**. And by the way this kind of attack is *prevented* if you choose to use `SecureDexClassLoader` as explained later on in this guide.

The object `mDexClassLoader` is then initialized as a `DexClassLoader` instance, which loads all the classes into *examplejar* and caches their optimized version into `dexOutputDir`. No native library is included since the third parameter of the constructor is `null` and the ClassLoader of the current activity is passed as parent class loader.

Finally the designated class is, at first, loaded by invoking the `loadClass()` method on `mDexClassLoader` with the **full class name** provided as a parameter and, secondly, instantiated through the `newInstance()` method and the forced casting to `MyClass`. The three different **catch blocks** are used to handle different exceptions that may be raised during the process.

Package Name In Java every class is associated to a **package name**. A **package** is a *grouping of related classes, interfaces and enumerations* providing **access protection** and **name space management**. In particular in *Grab'n Run* packages names are accepted if and only if they are *a sequence of at least two not-empty, dot-separated words, which starts and ends with a word, not with a dot*. This implies that the following are all examples of **invalid** package names: `com`, `.com.application`, `com..application`, `com.application.`, while **suitable** package names are `com.application` or `it.polimi.necst.gnr`. As you will see later on in this tutorial, package names perform a **relevant functionality** in **GNR** system since they *link containers to be verified with the certificate used to do so*.

> **Warning:** Notice that a **full class name** is required to successfully load a class and so the **complete package name** separated by dots must **precede** the **class name**. Referred to the example, full class name is `com.example.MyClass` and not just the short class name `MyClass`, which would produce a failure in the class loading operation. In particular if it is the case that a short class name is provided in stead of a full one, it is likely that a `ClassNotFoundException` will be raised at runtime.

This snippet of code is perfectly fine and working but it is **not completely secure** since neither integrity on the container of the classes, neither authentication on the developer of the container are checked before executing the code. And here comes `SecureDexClassLoader` to solve these issues.

1.2.2 Using SecureDexClassLoader to load dynamic code securely

In order to improve the security of the snippet of code shown in *Using standard DexClassLoader to load code dynamically* a new version of the code is presented through the use of `SecureDexClassLoader` and `SecureLoaderFactory`.

At first you should create a `SecureLoaderFactory` object as shown here:

```
SecureLoaderFactory mSecureLoaderFactory = new SecureLoaderFactory(this);
```

This is an helper class necessary to generate a `SecureDexClassLoader` object. But before performing this step you have to initialize and provide to `mSecureLoaderFactory` an **associative map** which links all the package names of the classes that you want to dynamically load to one *developer certificate*, which is stored at a **secure web location** (i.e. an HTTPS link) and which was previously used to sign the *jar* or *apk* container which holds those classes.

Developer Certificate a certificate, which in Android can be even *self-signed*, used to sign all the entries contained in a *jar* or in an *apk* container. Notice that in the Android environment in order to run an application on a smart phone or to publish it on a store, the *signing step* is **mandatory** and can be used to check that an *apk* was actually written and approved by the issuer of the certificate. For more details on signing applications and certificate, please check here.

So in this example we assume that all the classes belonging to the package `com.example` have been signed with a self-signed certificate, stored at `https://something.com/example_cert.pem`. Since here you just want to load `com.example.MyClass` the following snippet of code is enough:

```
Map<String, URL> packageNamesToCertMap = new HashMap<String, URL>();
try {
        packageNamesToCertMap.put(      "com.example",
                                        new URL("https://something.com/example_cert.pem"));

} catch (MalformedURLException e) {
        // The previous URL used for the packageNamesToCertMap entry was a malformed one.
        Log.e("Error", "A malformed URL was provided for a remote certificate location");
}
```

Note: Any *self-signed certificate* can be used to validate classes to load as long as it is not expired and it suits the standard X509 Certificate format. The only exception is represented by the **Android Debug Certificate**, a certificate used to sign applications before running them in debug mode and not safe to use during production phase. `SecureDexClassLoader` has been instructed to automatically reject class loading for classes whose package name has been associated for signature verification to the **Android Debug Certificate** and so **DO NOT USE IT** to check the signature of your containers.

Note: You may want to insert more than one entry into the associative map. This is useful whenever you want to use the same `SecureDexClassLoader` to load classes which belong to different packages. Still remember that each package name can only be associated with **one and only one** certificate location. Pushing into the associative map an entry with an already existing package name will simply overwrite the previously chosen location of the certificate for that package name.

Warning: For each entry of the map only an **HTTPS** link will be accepted. This is necessary in order to **avoid MITM (Man-In-The-Middle)** attacks while retrieving the *trusted* certificate. In case that an **HTTP** link is inserted, `SecureLoaderFactory` will enforce *HTTPS protocol* on it and in any case whenever no certificate is found at the provided URL, no dynamic class loading will succeed for any class of the related package so **take care to verify** that certificate URL is correctly spelled and working via **HTTPS** protocol.

Now it comes the time to initialize a `SecureDexClassLoader` object through the method `createDexClassLoader()` of `SecureLoaderFactory`:

```
SecureDexClassLoader mSecureDexClassLoader =
        mSecureLoaderFactory.createDexClassLoader(      jarContainerPath,
                                                        null,
                                                        getClass().getClassLoader(),
                                                        packageNamesToCertMap);
```

`mSecureDexClassLoader` will be able to load the classes whose container path is listed in `jarContainerPath` and it will use the `packageNamesToCertMap` to retrieve all the required certificate from the web and import them into an application private certificate folder. Also notice that in this case no directory to cache output classes is needed since `SecureDexClassLoader` will automatically reserve such a folder.

Warning: As stated in the API documentation `jarContainerPath` may link many *different containers* separated by `:` and for such a reason the **developer is responsible** of filling the associative map of the certificates location accordingly with all the entries needed to cover all the package names of the classes to be loaded.

Note: `DexClassLoader`, the standard class from Android API, is able to parse and import only those *jar* and *apk* containers listed in `jarContainerPath` which are directly saved on the mobile device storage. In addition to this `SecureDexClassLoader` is also capable of **downloading remote containers** from the web (i.e. **HTTP or HTTPS URL**) and to import them into an application-private directory to avoid code injections from attackers.

Example:

```
jarContainerPath = "http://something.com/dev/exampleJar.jar";
```

This `jarContainerPath` will retrieve no resource when used in the constructor of `DexClassLoader` but it is perfectly fine as a first parameter of the `mSecureLoaderFactory.createDexClassLoader()` call, as long as a *jar* container is actually stored at the remote location.

Finally you can use the resulting `mSecureDexClassLoader` to load the desired class in a similar fashion to `DexClassLoader`:

```
try {
        Class<?> loadedClass = mSecureDexClassLoader.loadClass("com.example.MyClass");

        // Check whether the signature verification process succeeds
        if (loadedClass == null) {

                // One of the security constraints was violated so no class
                // loading was allowed..
        }
        else {

                // Class loading was successful and performed in a safe way.
                myClassInstance = (MyClass) loadedClass.newInstance();

                // Do something with the loaded object myClassInstance
                // i.e. myClassInstance.doSomething();
        }

} catch (ClassNotFoundException e) {
        // This exception will be raised when the container of the target class
        // is genuine but this class file is missing..
        e.printStackTrace();
} catch (InstantiationException e) {
        e.printStackTrace();
} catch (IllegalAccessException e) {
        e.printStackTrace();
}
```

It is important to notice that, differently from `DexClassLoader`, the `mSecureDexClassLoader.loadClass()` call will return `null` whenever **at least one of the following security constraints is violated**:

- The *package name* of the class used as a parameter of `loadClass()` was **not previously included in the associative map** and so it do not exist any certificate that could be used to validate this class.
- The *package name* of the class used as a parameter of `loadClass()` was previously included in the associative map but the **related certificate** was **not found** (URL with no certificate file attached or no connectivity) or **not valid** (i.e. expired certificate, use of the Android Debug Certificate).
- The *container file* of the required class was **not signed**.
- The *container file* of the required class was **not signed with the certificate associated** to the package name of

the class. [Missing trusted certificate]

- At least one of the **entry** of the *container file* do **not match its signature** even if the certificate used to sign the container file is the trusted one. [Possibility of **repackaged container**]

For all of these reasons you should always check and pay attention when a **null** pointer is returned after a `mSecureDexClassLoader.loadClass()` call since this is a clear clue to establish either a wrong set up of `SecureLoaderFactoty` and `SecureDexClassLoader` or a security violation. *Informative and debug messages* will be generated in the logs by the classes of the Grab'n Run library in order to help you figure out what it is happening.

Note: Every time that `SecureDexClassLoader` finds out a (possibly repackaged) **invalid container**, it will immediately **delete** this file from its **application-private directory**. Nevertheless if this container is *stored on your device* it may be a good idea for you, as a developer, after having double checked that you have properly set up `SecureDexClassLoader`, to **look for a fresh copy** of the container or at least **not to trust** and delete this container from the phone.

Please notice, on the other hand, that the three exceptions caught in the try-catch block surrounding the `loadClass()` method behaves and are thrown in the same way as it would happen with `DexClassLoader`.

Finally for clarity the **full snippet of code** presented in this section is reported here:

```
MyClass myClassInstance = null;
jarContainerPath = "http://something.com/dev/exampleJar.jar";

try {
        Map<String, URL> packageNamesToCertMap = new HashMap<String, URL>();
        packageNamesToCertMap.put(      "com.example",
                                        new URL("https://something.com/example_cert.pem"));

        SecureLoaderFactory mSecureLoaderFactory = new SecureLoaderFactory(this);
        SecureDexClassLoader mSecureDexClassLoader =
                mSecureLoaderFactory.createDexClassLoader(      jarContainerPath,
                                                                null,
                                                                getClass().getClassLoader(),
                                                                packageNamesToCertMap);

        Class<?> loadedClass = mSecureDexClassLoader.loadClass("com.example.MyClass");

        // Check whether the signature verification process succeeds
        if (loadedClass == null) {

                // One of the security constraints was violated so no class
                // loading was allowed..
        }
        else {

                // Class loading was successful and performed in a safe way.
                myClassInstance = (MyClass) loadedClass.newInstance();

                // Do something with the loaded object myClassInstance
                // i.e. myClassInstance.doSomething();
        }

} catch (ClassNotFoundException e) {
        // This exception will be raised when the container of the target class
        // is genuine but this class file is missing..
        e.printStackTrace();
```

```
} catch (InstantiationException e) {
        e.printStackTrace();
} catch (IllegalAccessException e) {
        e.printStackTrace();
} catch (MalformedURLException e) {
        // The previous URL used for the packageNamesToCertMap entry was a malformed one.
        Log.e("Error", "A malformed URL was provided for a remote certificate location");
}
```

1.2.3 Wiping out cached containers and certificates

In order to *improve performance* and offer the possibility to *partially work also when connectivity is limited*, `SecureDexClassLoader` will store certificates retrieved from the web and all containers into specific **application-private directories**.

Every time that a **resource** (container or certificate) is needed to load or verify a class, `SecureDexClassLoader` will at first look for it inside its private directories and then, if no match is found, possibly attempt to download it from the web or found it at a specified location on the device (this last option is applicable only for containers).

Even if these **caching features** may come really useful and *speed up* significantly `SecureDexClassLoader` execution, it would be also nice for the developer to have the possibility to **choose** whether a **fresh or cached copy** of either a certificate or a container should be used for the *dynamic loading operations*. And that is the reason why `SecureDexClassLoader` provides a method called `wipeOutPrivateAppCachedData()` to manage this choice.

To present this method let us consider again the previous scenario shown in *Using SecureDexClassLoader to load dynamic code securely*: after having tried to load `com.example.MyClass`, if you want to *delete both the cached certificates and the containers* used by the related `mSecureDexClassLoader`, in order to impose for the next loading operation the retrieval of **fresh resources**, the call to perform is the following:

```
mSecureDexClassLoader.wipeOutPrivateAppCachedData(true, true);
```

Warning: After that you *have erased at least one cached resource between the certificates and the containers*, `mSecureDexClassLoader` will always return `null` for **consistency reason** to any invocation of the `loadClass()` method. So it will be **necessary** for you to require a **new** `SecureDexClassLoader` instance to `SecureLoaderFactory` through the invocation of the `createDexClassLoader()` method before being able to dynamically and securely load other classes.

CHAPTER 2

Why should I use Grab'n Run?

A significant question that every developer faces every time that (s)he adds a new library to his/her project is the following: "*Is it worthy to add this library? Does it provide something really necessary to my project?*"

In this specific case those two questions can be reformulated as follows: "*Aren't Android API already able to handle dynamic class loading? Why should I use for example* `SecureDexClassLoader` *in stead of the regular* `DexClassLoader` *? Does it really enhance the standard class with something relevant?*".

To answer all of these questions let us consider the case of choosing `SecureDexClassLoader`, one of the classes provided by **Grab'n Run**, in stead of DexClassLoader, the standard class provided by the Android API, to *dynamically load* **.dex classes** into your Android application at run time.

First of all `SecureDexClassLoader` provides a couple of slight but significant **improvements** over the standard class in terms of **functionalities**:

- `SecureDexClassLoader` lets you retrieve *dynamically* also classes from *.jar* and *.apk* containers which are **not located directly on the phone** running the application as long as you simply provide a **valid remote URL** for the resource, while `DexClassLoader` is only able to cache classes from containers stored on the phone.
- `SecureDexClassLoader` is a **thread-safe** library and so, after that you have created your `SecureDexClassLoader` instance on the main thread of your application, you can launch different threads, each one performing dynamic class loading on the very same `SecureDexClassLoader` instance **without** incurring in nasty **race conditions** and **concurrencies exceptions**.

In addition and above all `SecureDexClassLoader` ensures relevant **security features** on classes that you dynamically load that are not possible to check or implemented with `DexClassLoader` like:

- **Fetch remote code in a secure way**: `SecureDexClassLoader` retrieves remote code either via HTTP or HTTPS protocol. In both cases it **always verifies** and validates the downloaded containers **before actually loading classes** inside of them.
- **Store containers in secure application-private locations**: *SecureDexClassLoader'* also **prevents** your application from being a possible target of **code injection attacks**. This attack becomes feasible whenever you use the standard `DexClassLoader` and you provide as an optimized cache folder for *dex* files a directory which is located in a **world writable area** of your phone (i.e. external storage) or you decide to load classes from a container which is, once again, stored in a world writable folder. `SecureDexClassLoader` on the other hand manages this situation for you by choosing *application-private directories* for caching *dex* files and storing containers and certificates. This strategy represents an **effective** way to make the **attack infeasible**.
- **Developer authentication**: for each package containing classes to be loaded dynamically it is possible to ensure authentication of the developer who coded those classes though a *check of the signature on the container* of the classes against the *certificate of the developer* (which could possibly be even *self-signed*). **Non-signed** classes or those who were not signed by that required certificate will be rejected and **prevented from being loaded**.

- **Integrity**: during the *signature verification* process whenever one of the entries inside the container results to be *incorrectly signed*, `SecureDexClassLoader` recognized a **possible tampered or repackaged container** and it prevents your application from running the code of any classes inside this invalid and possible malicious container.

And these improvements come with a **convenient overhead** on the **performance** :)

This is possible since `SecureDexClassLoader` was implemented with an accurate **caching system** that, from one side, *prevents* your application **from continuously downloading** the same *jar* and *apk* **containers and certificates** and, from the other side, **avoid** it from **verifying every time** *the signature and the integrity* of already **checked containers**.

Moreover, for even *more performance concerned developers*, it is also possible to set the **strategy** which is going to be used by `SecureDexClassLoader` to **validate classes** before attempting to load them. In particular **two** options are provided:

1. **Lazy Strategy**: this mode implies that the **signature and integrity** of each container will be **evaluated only when** the `loadClass()` method will be invoked on *one of the classes*, whose package name is linked to this container. An ideal case of use for this mode is when you have quite a lot of containers and just a couple of classes to load, which may also vary from one execution to another and so validating all the containers in this case may be a waste of time.
2. **Eager Strategy**: in this mode the process of **signature and integrity** will be carried out on **all** the containers **immediately and concurrently** before returning an instance of `SecureDexClassLoader`. This choice implies that you will have to pay an **initial penalty (but still reduced since the verification process is driven concurrently among the containers)** on time of execution but then the time required for a `loadClass()` operation becomes almost equal to the corresponding operation performed with standard `DexClassLoader`.

By default eager strategy is applied but developers can *pick* the *lazy version* by adding a final `true` attribute to the `createDexClassLoader()` method invocation in `SecureLoaderFactory`. An example of use is shown in the following snippet of code (this is just a *slight modification* of one of the calls that you may have seen in Quick start and tutorial):

```
SecureDexClassLoader mSecureDexClassLoader =
        mSecureLoaderFactory.createDexClassLoader(      jarContainerPath,
                                                        null,
                                                        getClass().getClassLoader(),
                                                        packageNamesToCertMap,
                                                        true);
```

CHAPTER 3

Discussion of an example project

Before digging into this section, you are **strongly** encouraged to read Quick start and tutorial for an **introductory description** on the features of Grab'n Run library.

The **aim of the sample application** is to give you some *hints on how to use the classes in Grab'n Run and how they will behave across different contexts*. The **source code** of the example can be found in the `example` folder of *Grab'n Run* repository.

Different extracts of code will be considered and explained in the following sections of this page but before analyzing the code it may be convenient to retrieve it and to set up an **already prepared Android smart phone emulator** that contains all the containers needed to run the example code..

3.1 Retrieve the example code and the emulator

3.1.1 Retrieve Grab'n Run full repository

At first you will need to recover *Grab'n Run* example code. In order to do so you need to have **Git** installed on your machine. The latest version can be found at Git download page.

Next open a terminal and **clone** the example repository into `grab-n-run`, a local folder located at `absolute_path_to_gnr_repo`, through Git:

```
$ cd <absolute_path_to_gnr_repo>
$ mkdir grab-n-run
$ cd grab-n-run
$ git clone "https://github.com/lukeFalsina/Grab-n-Run.git"
```

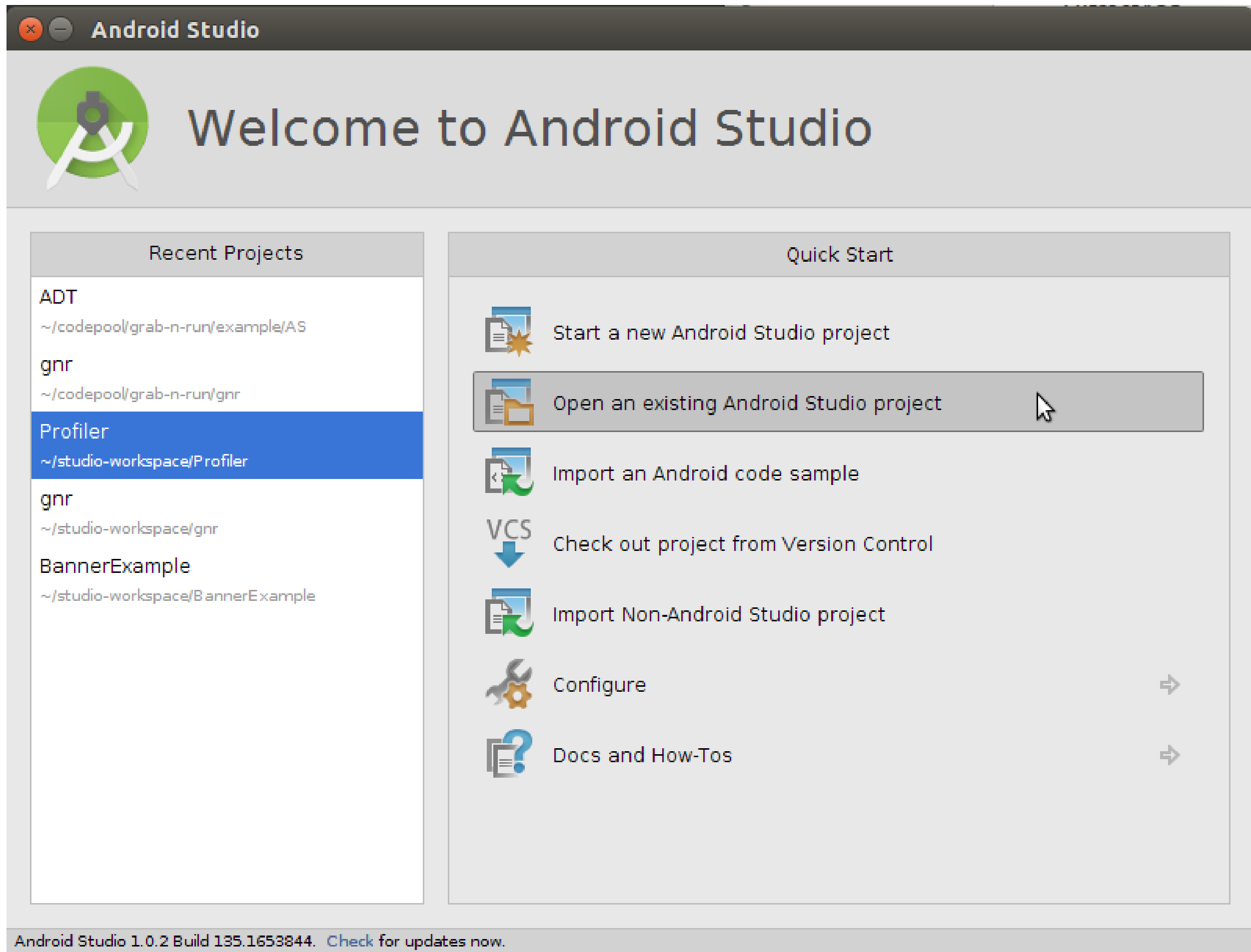
At the end of the process you will have all the GNR code locally including a copy of the *example application* and of the *documentation*.

3.1.2 Include Grab'n Run example code in your IDE

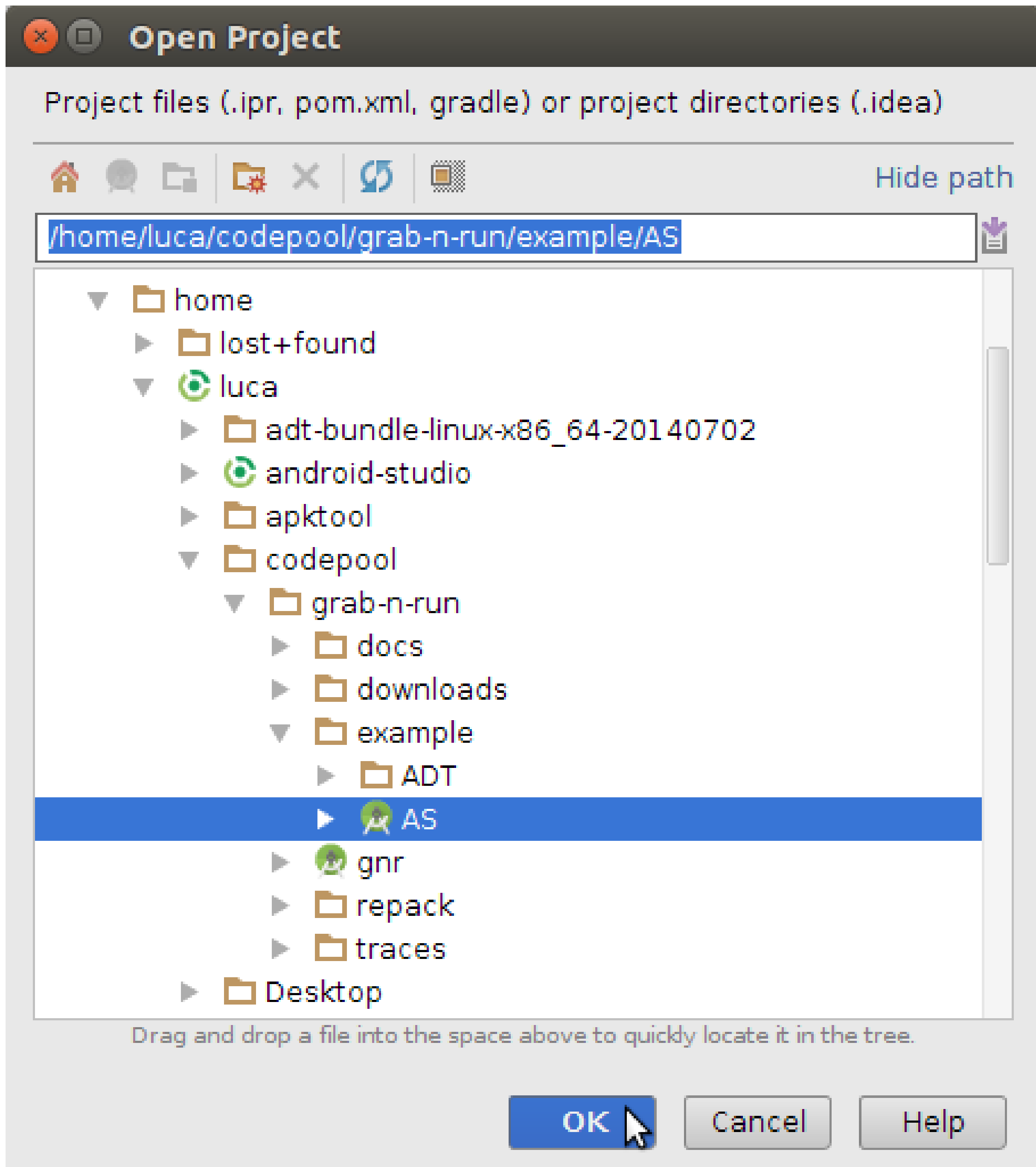
The next step is *importing the example sources* into an **IDE**. The process will be now described for both **Android Studio (AS)** and **Android Development Tool (ADT)**.

1. **Android Studio (AS)**

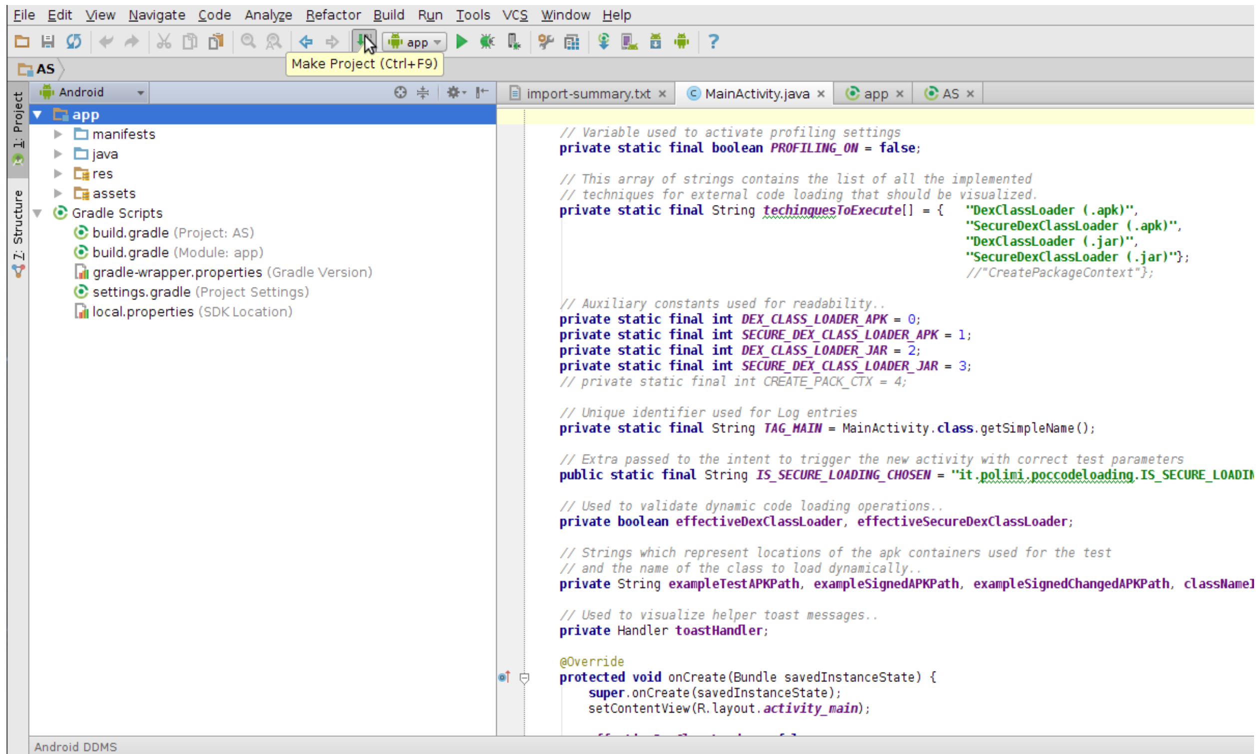
In the welcome window of Android Studio select from the *Quick Start* menu the option "Open an existing Android Studio project".



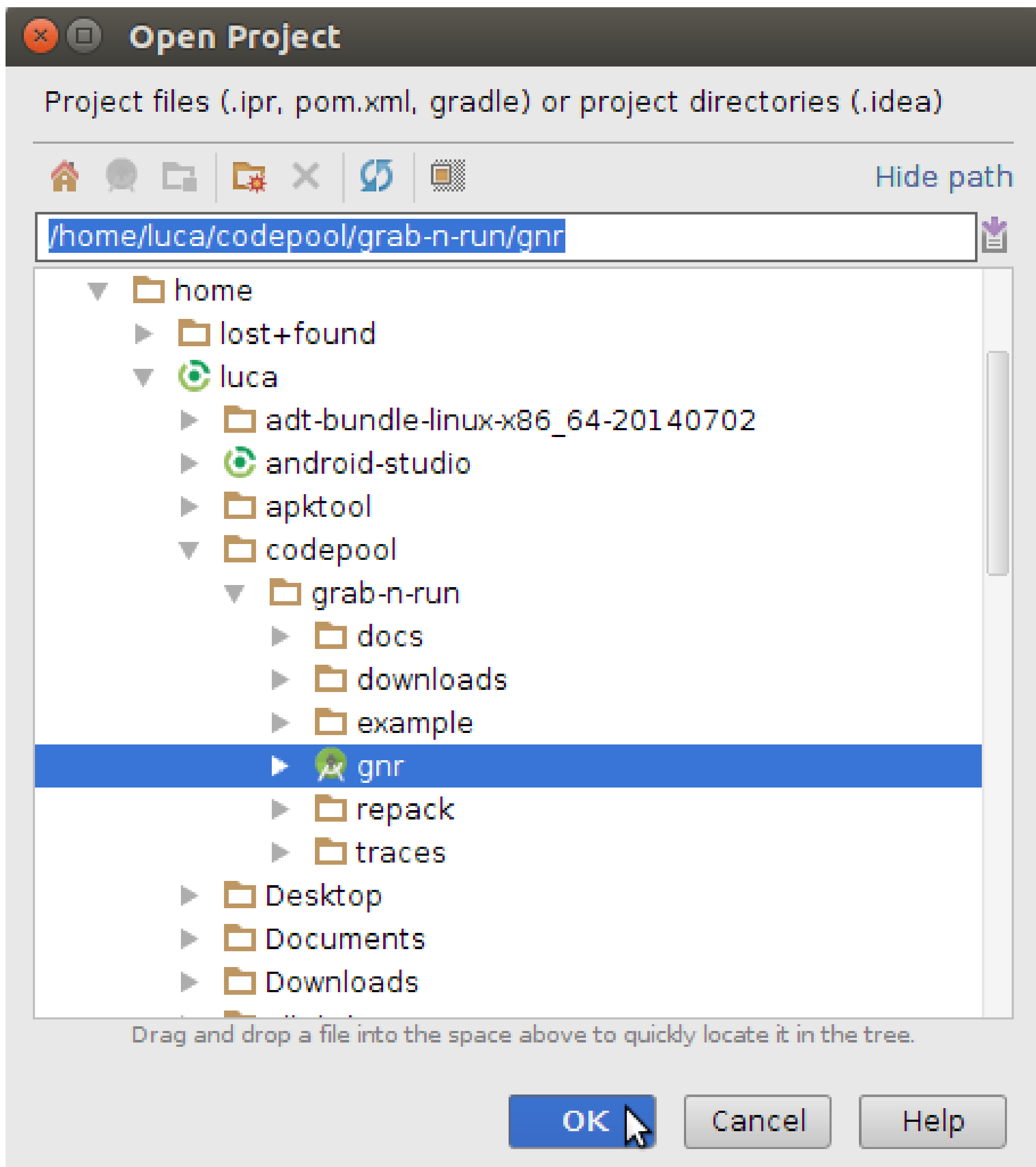
Next navigate to the `grab-n-run` folder in which you previously cloned the repository and then pick the project `AS` from the `example` subfolder as shown in the image below.



The example project should have been now successfully imported! It may be necessary to rebulit it again by picking the option "Make Project".

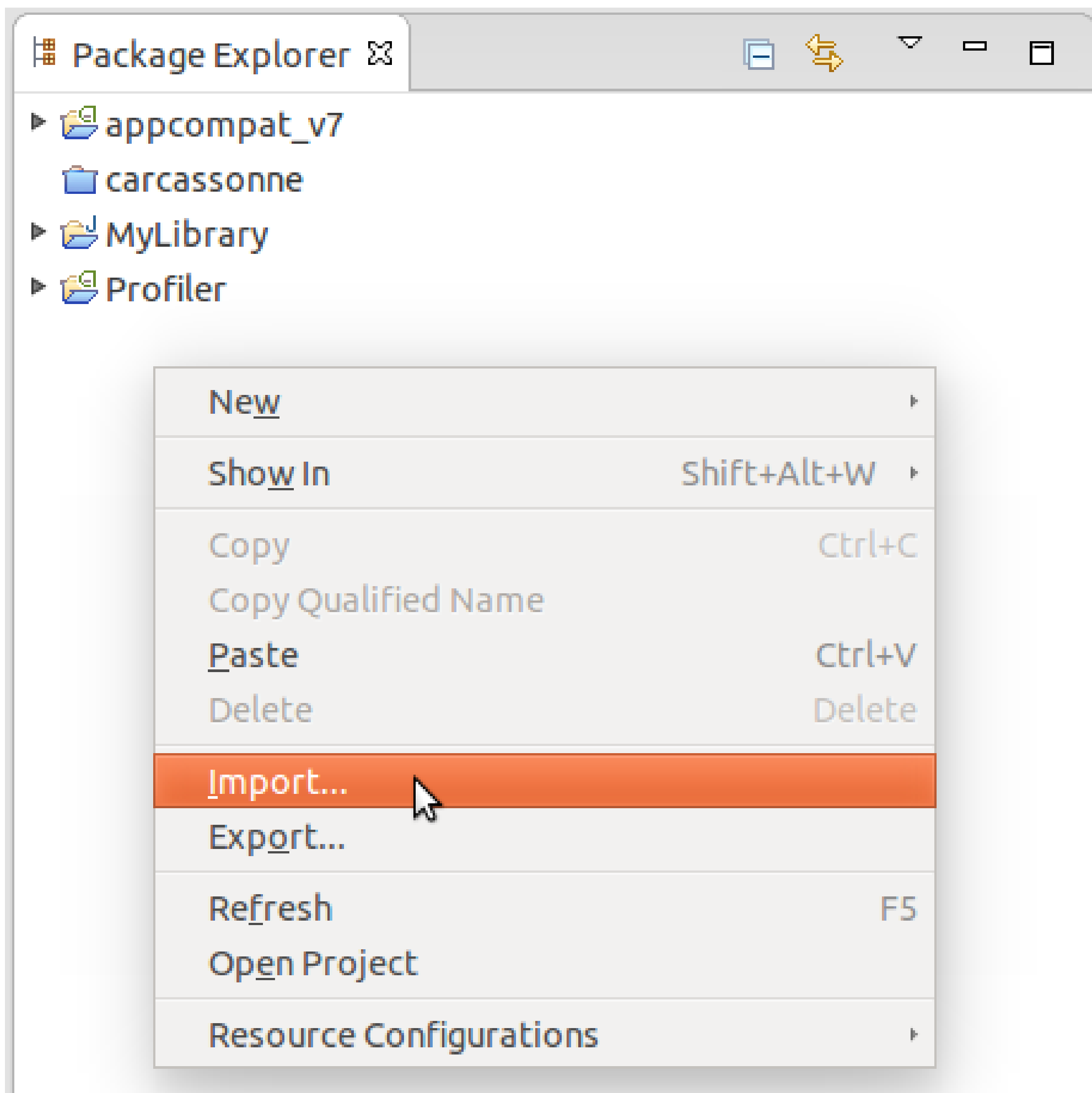


P.S. Notice that you can open in *Android Studio* also the **original GNR library project** by using the very same procedure but by picking the `gnr` Studio project from the main `grab-n-run` folder in stead of the `example/AS` one.

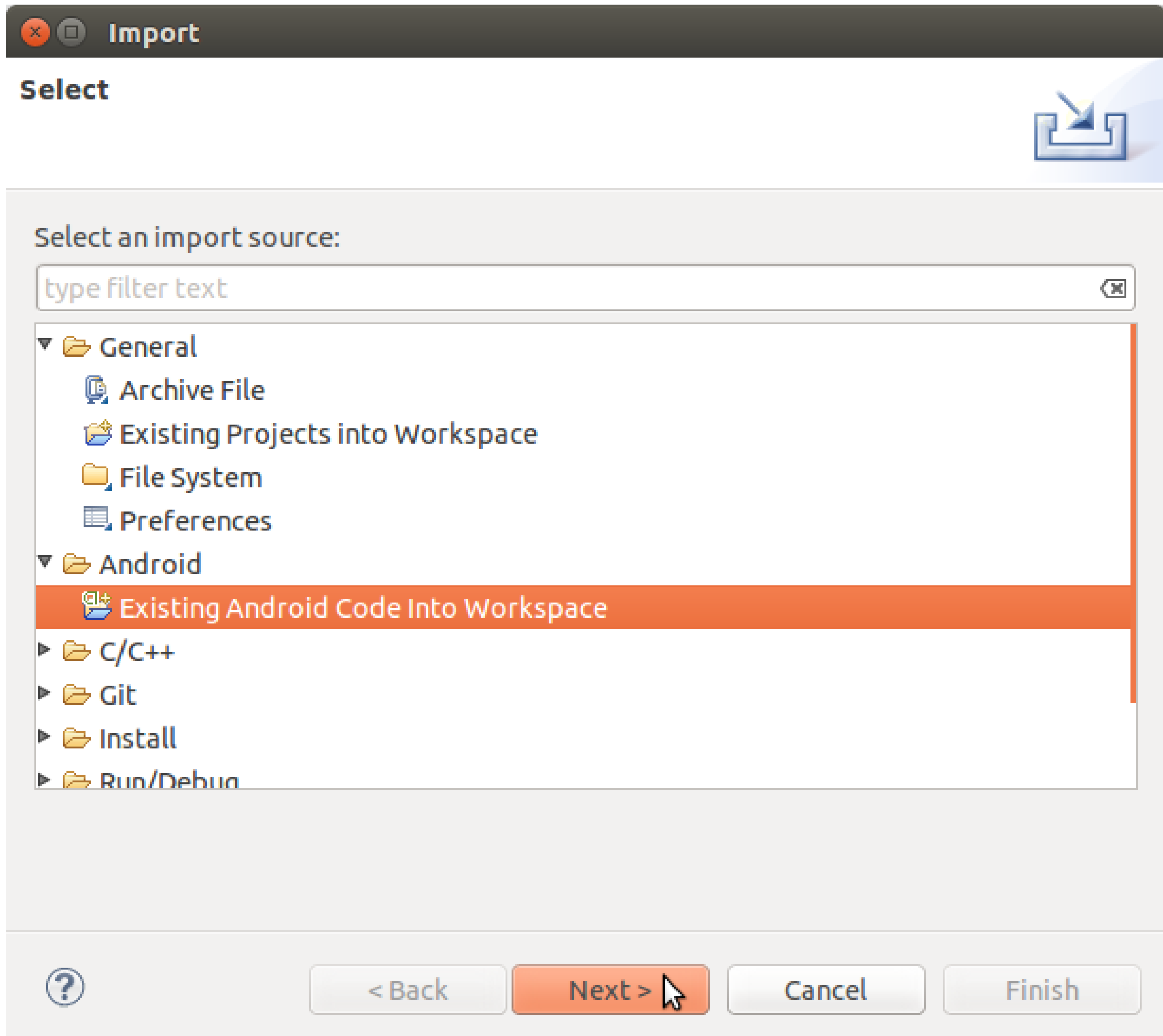


2. **Android Development Tool (ADT)**

At first right click in the *Package Explorer* and select "Import.."



Next select under the *Android* folder “Existing Android Code Into Workspace” and then “Next >”



By pressing the "Browse..." button point the *Root Directory* to the `grab-n-run` folder in which you previously cloned the repository and then to the subfolder `grab-n-run/example/ADT`. You should be now able to the see and select the candidate project `ExampleAppGNR` (*an example application which makes use of GNR*). In the end press "Finish" to import the example project. Below you can see a screenshot which summarizes all the settings before the "Finish" button is clicked.

Import Projects

Select a directory to search for existing Android projects

Root Directory: /home/luca/codepool/grab-n-run/example/ADT Browse...

Projects:

Project to Import	New Project Name
/home/luca/codepool/grab-n-run/example/ADT	ExampleAppGNR

Select All

Deselect All

Refresh

Copy projects into workspace

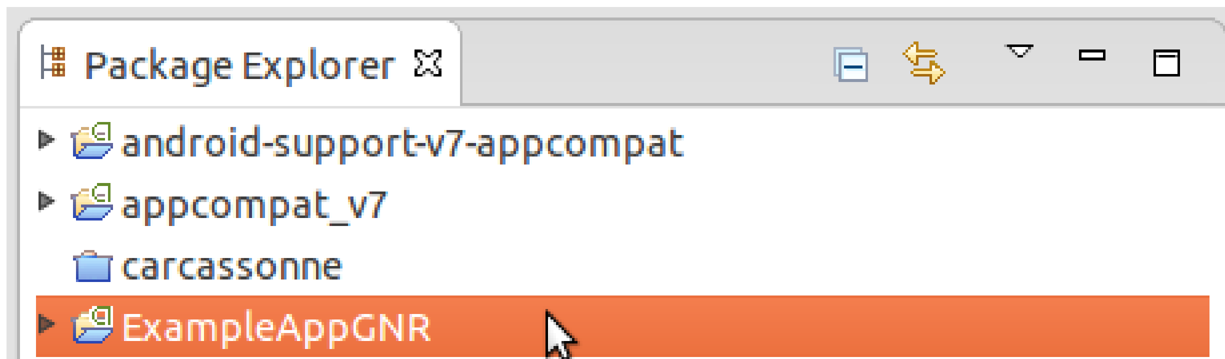
Working sets

Add project to working sets

Working sets: Select...

< Back Next > Cancel Finish

At the end of this process you should have been able to **correctly import** the example application!



3.1.3 Retrieve and set up the emulator

Then it is time to retrieve the **emulator** used to run the example application. You can easily find it in the `assets` folder of the `example` repository. So once that you have located the compressed file `ExampleAppGNREmu.tar.gz` containing the emulator, open a terminal and at first copy this file into your *home* folder:

```
$ cd <absolute_path_to_gnr_repo>/example/
$ cp ExampleAppGNREmu.tar.gz ~
```

Next decompress this container:

```
$ cd ~
$ tar xzf ExampleAppGNREmu.tar.gz
```

This operation will generate two files, a folder called `ExampleAppGNREmu.avd` and a configuration file `ExampleAppGNREmu.ini`. In the end move these two files into the Android emulator folder, normally located at `/home/<your_username>/.android/avd`:

```
$ mv ExampleAppGNREmu.avd ExampleAppGNREmu.ini .android/avd
```

The last step consist in editing the `path` variable stored in the configuration file. So open `ExampleAppGNREmu.ini` at the final location with a text editor and change the path variable in order to match the current location of the `ExampleAppGNREmu.avd` folder. So if my user name is for example *bill90*, I need to change the path variable from `path=/home/<USER_NAME>/.android/avd/ExampleAppGNREmu.avd` to `path=/home/bill90/.android/avd/ExampleAppGNREmu.avd`.

Before starting the emulator in your **IDE**, remember to **verify that the SDK version 17** is installed on your machine since the emulator targets that version. Otherwise you can *also edit the emulator configuration* from your IDE to target a different and **more recent** version of the SDK which is installed on your machine.

Note: Android emulator is unfortunately pretty slow and requires also a big bunch of resources and that is the reason why it may be not supported by different machines. A couple of empirical suggestions in this direction are the following:

- If possible, try to target directly **SDK version 17**, as it results to me that the more recent SDK version you target, the more time the emulator requires before setting up.
- It is a really good idea to enable the **snapshot feature**. This lets the system frame the current situation of the emulator when you turn it off and load it back whenever you restart the emulator with a *significant reduction of the waiting time*. This post explains how to enable the feature.
- Emulator can be switched between landscape and portrait view by pressing `ctrl + F12`. This can be useful to interact properly with the example application.

When the emulator is finally set up, you can start it in either **ADT Eclipse** or **Android Studio** (it may take time depending on your machine..). Next, whenever you want to run the example code and the IDE asks which device should be used, remember to **select this emulator as the running Android device**.

In case you need to integrate this previous concise walk-through, please give a look at these other resources:

- https://blahti.wordpress.com/2011/08/24/how-to-export-and-import-android-virtual-device-avd-files/
- http://stackoverflow.com/questions/4575167/android-how-to-copy-the-emulator-to-a-friend-for-testing

3.2 List of example containers

In order to understand correctly the following detailed discussion, it is fundamental to first introduce the containers (*jar* and *apk* archives), retrieved for the code loading in the example code. Here is a list of the string variables that store the path to various containers:

- `exampleSignedAPKPath`: URI of a **benign** toy *apk* container signed with a valid *developer certificate*.
- `exampleTestAPKPath`: path location pointing to the same **benign** *apk* container but this time signed with the *Android Debug Certificate*.
- `exampleSignedChangedAPKPath`: URI pointing to a **handled version** of the same container stored at `exampleSignedAPKPath` in which a part of the signatures has been modified.
- `jarContainerPath`: path location to the **benign** *jar* container used to customize the view elements inside an example activity.
- `jarContainerRepackPath`: URI pointing to a **malicious repackaged** version of the original container stored at `jarContainerPath`.

3.3 MainActivity.java

MainActivity is the **entry point** of the sample application. In its overloaded method `onCreate()` it initializes through a `ListView` a set of buttons used to select the *different test cases* present in the application.

3.4 DexClassLoader (apk) vs SecureDexClassLoader (apk)

In this first scenario you will consider how to retrieve an Activity class, whose name is `NasaDailyImage`, stored in the *apk* container, called *test.apk*, through the use of DexClassLoader and `SecureDexClassLoader`.

The relevant **code** in this case is the one of the two methods `setUpDexClassLoader()` and `setUpSecureDexClassLoader()`, which are triggered by tapping the related two buttons on the `MainActivity` view.

3.4.1 setUpDexClassLoader()

In this method a standard initialization of a `DexClassLoader` is applied. So at first the usual **application-private, writable directory** for caching loaded *.dex* classes must be set up.

Then a `DexClassLoader` object is initialized using *test.apk*, a container located directly in the phone external storage (as described by `exampleTestAPKPath`), as its *jar path* for the classes to load.

Finally the `NasaDailyImage` Activity is loaded. If such an operation is successful the **simple name** of the **loaded class** is shown to the user through a *toast message*; otherwise different **exceptions** are raised and show again through a toast message an appropriate helper message.

3.4.2 setUpSecureDexClassLoader()

In this method **repeated** `loadClass()` **calls** are performed on differently initialized `SecureDexClassLoader` instances in order to *show different behaviors* of the loader class while retrieving the usual `NasaDailyImage` Activity.

At first a `SecureLoaderFactory` object is created. Then this instance is used to generate three `SecureDexClassLoader` that covers different cases and ends up with different results on the load operation:

1. **Test case 1:** Load a class through `SecureDexClassLoader` without providing an associative map for certificates location

   This first test case shows a **possible error** that a developer may encounter when using this library for the first time. If you want to have the location of the certificate being computed by reversing the package name, as explained in *Reverse package name to obtain remote certificate URL*, you still need to **populate an associative map** with entries like (*"any.package.name"*, **null**) and use it as a parameter of the method `createDexClassLoader()`. To understand why the class works in this way think of this system as a kind of white listing. Only those classes inside package names which are *declared into the associative map* or *directly descend* from one of the declared package names will be considered as possible valid ones, while all classes belonging to a **not listed package name or not a descendant of the declared ones** will be **immediately rejected**.

   And this is exactly what happens in this test case where **no associative map is provided** and so all the classes in the two containers, including the target `NasaDailyImage`, are **prevented from being loaded** since there is *no clue on the certificate location*.

2. **Test case 2:** Unsuccessful load of a class through `SecureDexClassLoader` with an associative map *(Debug certificate)*

   In the second test case you can see different ways to **populate** the associative map `packageNamesToCertMap`, used to *link packages with certificates location*.

   > **Warning:** Always keep in mind that **prior** to **downloading** a certificate from the **web** the certificate for that package will be **searched inside the application-private directory** reserved for certificates and then possibly at the remote location. If you wish to *just look at the remote URL* without considering cached certificates, always remember to **wipe out private application data** through the invocation of the method `wipeOutPrivateAppCachedData()` **before dismissing** your `SecureDexClassLoader` instances. In such a way every time that a new `SecureDexClassLoader` is created, you will be sure that no cached resource will be associated with it.

   The first `put()` *call* inserts the package name *headfirstlab.nasadailyimage* of the class that we would like to load later in the example and associates it with a **valid remote URL**. What you can immediately notice by pointing your browser to that URL is that the *remote certificate* in this case is a **self-signed developer** one since the **subject** of the certificate is **also** the **issuer** of it but, as it is mentioned in the Quick start and tutorial, this is perfectly fine in the **Android** environment.

   The *second entry* inserted into the associative map provides a *remote URL* to an **inexistent certificate** (once again you can try to point there your browser to easy spot this out). More over since *no certificate for the package name ''it.polimi.example'' has been already cached into the application-private certificate directory*, then **no certificate** is **available** for it and that is the reason why *any class* belonging to the `it.polimi.example` package will be **rejected and prevented from being loaded** by `SecureDexClassLoader`.

   Lastly the third `put()` call on the associative map will insert a package name that will be also used to *construct the remote certificate URL* (**reverse package name**). Once again the final remote URL (`https://polimi.it/example3/certificate.pem`) points to no certificate so any class, whose package name is *it.polimi.example3*, will be rejected from being loaded.

   In the end a `SecureDexClassLoader` is generated using as a container file a valid *apk* containing the target class but **signed with a certificate**, the *Debug Android Certificate*, which is different from the one issued by the developer. For such a reason the result of the `loadClass()` method will be that *no class object is going to be returned* since the apk is **not signed** with the **required certificate**.

3. **Test case 3:** Unsuccessful load of a class through `SecureDexClassLoader` with an associative map *(Failed signatures verification of some container's entries)*

In the third test case you can immediately notice that all the settings for the invocation of `SecureDexClassLoader` are equals to those of the previous case except for the chosen *apk* container. In fact, while before the container was signed with a non valid certificate, this time the container is signed with the **right certificate** but someone **modified** a couple of the **entries signature**, which do not match anymore with the one obtained during the signing procedure. To sum up also in this case *no class will be loaded* since this container results to be **partially corrupted** and so not safe.

4. **Test case 4:** Successful load of a class through `SecureDexClassLoader` with an associative map

   In this last test case a **successful example** of dynamic code loading is shown. This time `SecureDexClassLoader` is initialized with a **valid** *apk* container, **signed** with the **correct developer certificate**, and with the associative map previously initialized in *Test case 2*. The whole process works fine since this associative map contains the necessary key entry *headfirstlab.nasadailyimage* and the related developer **certificate** has been **already cached** during *Test case 2*. Finally during the **signature verification step** inside the `loadClass()` method all the entries inside the container match properly with their signature and the certificate used for that signing process is exactly the one linked to *headfirstlab.nasadailyimage* package. That is the reason why *dynamic loading* of `NasaDailyImage` activity is **allowed**.

3.5 DexClassLoader (jar) vs SecureDexClassLoader (jar)

A different scenario to show the power of *dynamic code* loading and the **security weakness** of the standard `DexClassLoader` is represented by the following example. In this case another activity (the source code is contained into *DexClassSampleActivity.java*) instantiates a certain number of **GUI components** (a couple of buttons, a text view, a switch..) and then **customizes** them according to the methods of an object belonging to the **external** class `ComponentModifier`, which is **dynamically loaded** at run time.

Depending on the user choice (tapping the first button in stead of the second one) a different extension class of `ComponentModifier` is loaded and a different behavior is shown to the user even if the static code shown in `DexClassSampleActivity` is exactly the same (as you can easily check by inspecting the method `onBtnClick()`). This loading operation can be realized easily by means of `DexClassLoader` as shown in the method `retrieveComponentModifier()` of the source code..

That's just a pity that the container used to load dynamically the class by `DexClassLoader` in this example is actually *randomly selected at run time* between either a benign version or a **repackaged one** of the original *apk* and so **malicious code** could potentially have been **executed** *without the user even notice it*!

But let's explain how this could possibly happen: in `DexClassSampleActivity` there is a simple private method called `randomContainerPathChoice()`, which in this case is invoked before the instantiation of both `DexClassLoader` and `SecureDexClassLoader` and which **select randomly the path** of either the **benign** version of the `ComponentModifier` container, stored in the string `jarContainerPath`, or the path of the **repackaged** one with the string `jarContainerRepackPath`.

`DexClassLoader` *won't notice and care* about this difference as long as in both the containers there is an implementation of the required **target class** to load and that is the reason why repeating tapping on the first button ''Click me!" in the Activity screen multiple times will end up in executing two different version of the same `FirstComponentModifierImpl` class.

On the other hand if you perform the same experiment with `SecureDexClassLoader` the repackaged *apk* container choice this time will be detected and blocked during the **signature verification procedure** against the developer certificate in the `loadClass()` method. This is possible since *malicious modified entries will not succeed in the signature verification check computed by considering both the initial signature stored inside the container and the developer certificate* retrieved from the associative map used to initialize the `SecureDexClassLoader` instance. Thanks to this, `SecureDexClassLoader` **won't load** the customization classes inside the *repackaged container* and it will just **end up the activity**, which is exactly the **secure** behavior that you, *as a developer*, would like to obtain :)

CHAPTER 4

Complementary topics

In the end of this documentation a couple of not so trivial use cases of *Grab'n Run* are presented. This section will not introduce new core concepts but it may help the developer to handle some **tricky situations**. For such a reason feel free to **skip this part** and eventually **come back later** to revise it whenever you will encounter one of the following situations while using the library.

4.1 Handle containers whose classes come from different package names which have a common relevant prefix

Before starting diving in this section it is important to recall the **relationship between package name and containers**.

Package name in apk containers *Apk* containers must contain just **one package name**, which must be chosen by the developer when a new application is created. The package name is then stored in the *Android Manifest* of the application. In order to have an application being admitted on the *Google Play* store, it is also fundamental that the chosen package name is **unique** and should **not change** for the whole life cycle of the application.

Package name in jar containers *Jar* containers on the other hand do **not** have such a **strict policy** as in *apk* containers. Hypothetically each class file contained in a *jar* archive may have a different package name and this mean that **many package names** can be present in the **same** *jar* container.

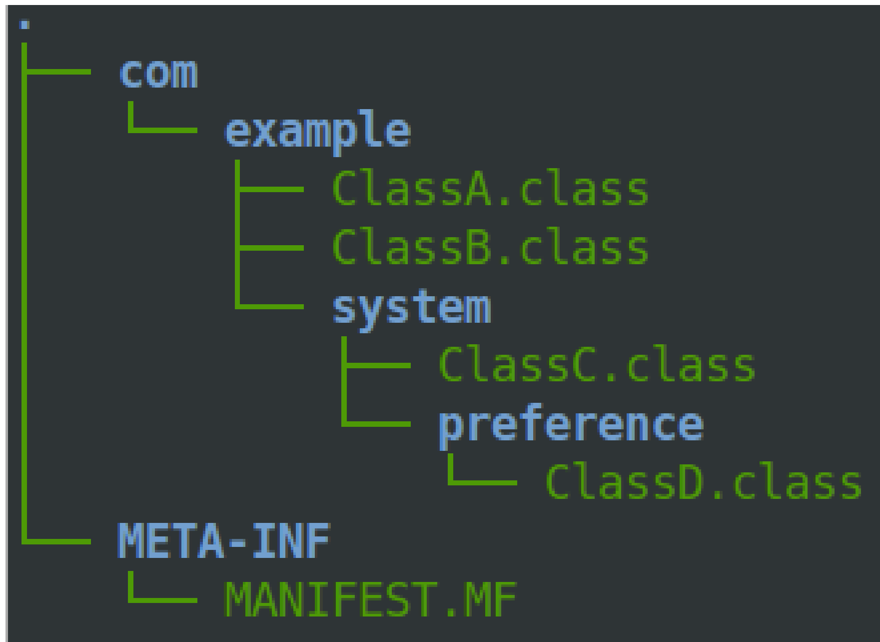
Common relevant prefix In *Grab'n Run* two package names share a relevant common prefix if their prefix match for at least two words separated by one dot.

Example: Consider the following package names:

1. `com.example.polimi`
2. `it.example.polimi`
3. `com.test`
4. `com.example.application.system`
5. `com.example.polimi.system`

- 1. and B. do **not** share any **common relevant prefix** since they differ in the initial word of the package name (`com` vs `it`).
- 1. and C. do **not** share any **common relevant prefix** since they just have one word of the package name in common (`com`).
- 1. and D. share a **common relevant prefix** (`com.example`).
- 1. and E. share a **common relevant prefix** (`com.example.polimi`).

Given these insights a first interesting situation to consider is when a developer wants to *load dynamically classes* from an external *jar* library which contains **more than one package name** that, anyway, share a **common relevant prefix**. Let us assume for example that the target library has the following structure:



In such a scenario we have four classes (`ClassA`, `ClassB`, `ClassC`, `ClassD`) which belongs to **three different packages**, whose names are respectively `com.example`, `com.example.system` and `com.example.system.preference`. Let use also assume that this container has being signed with a *valid self-signed certificate*, remotely located at `https://something.com/example_cert.pem`.

Questions now for the developer are:

1. *How should I fill in the associative map which links package names to remote certificate location in order to being able to load all the classes in this container?*
2. *Am I obliged to insert all three package names pointing to the very same certificate?*

Luckily the answer for the second question is **no**, which means that there is indeed an **easier way** to perform the job. *Grab'n Run* in fact was thought to make the whole dynamic class loading **secure but** at the same time **simple** for applications developers.

You can in fact handle this situation correctly by simply inserting into the associative map a **single entry** where the *key corresponds to the shortest among the package names* belonging to one of the classes that need to be loaded and the *value is the location of the remote certificate* used to sign the container. So in the **previous case** since the classes with the shortest package name are `com.example.ClassA` and `com.example.ClassB` the following code is appropriate to populate the map:

```
Map<String, URL> packageNamesToCertMap = new HashMap<String, URL>();

try {
        packageNamesToCertMap.put(      "com.example",
                                        new URL("https://something.com/example_cert.pem"));

} catch (MalformedURLException e) {

        // The previous entry for the map may not necessarily be the right one
        // but still it is not malformed so no exception should be raised.
        Log.e(TAG_MAIN, "A malformed URL was provided for a remote certificate location");
```

```
}
```

For the rest the developer may proceed as shown in *Using SecureDexClassLoader to load dynamic code securely*. The result will be that the container is going to be verified against the appropriate certificate and, if it is **genuine**, it will be *also possible to load the other two classes* in the archive with a **different package name** (`com.example.system.ClassC` and `com.example.system.preference.ClassD`).

4.2 Handle containers whose classes come from different package names with no relevant common prefix

Even if it is not such a common situation it is possible for a *jar* archive to *contain classes which belongs to different package names* and does not share any common relevant prefix. This situation, on the other hand, is **not practical** for *apk* containers since, in order to be **published** on Google Market, an application needs to have a **single** package name which more over must **not change** during its whole life cycle.

Anyway let us try to sketch the case of the previous cited jar archive and how to handle it with `SecureDexClassLoader`. As an example we can consider the scenario in which the goal is loading two classes, whose full class names are respectively `com.example.MyFirstClass` and `com.test.MySecondClass` and so which **differs** in the **package name** but are **both stored** in the **same container** `exampleJar.jar`. It is also supposed that this container has being signed with a *valid self-signed certificate*, remotely located at `https://something.com/example_cert.pem`.

In order to handle this situation correctly the developer is required to fill the **associative map** which links package names and certificates with **two entries**, one per each package name, which will *point to the same remote certificate*. This is exemplified in the following snippet of code:

```
Map<String, URL> packageNamesToCertMap = new HashMap<String, URL>();

try {
        packageNamesToCertMap.put(      "com.example",
                                        new URL("https://something.com/example_cert.pem"));
        packageNamesToCertMap.put(      "com.test",
                                        new URL("https://something.com/example_cert.pem"));

} catch (MalformedURLException e) {

        // The previous entries for the map may not be necessarily the right ones
        // but still they are not malformed so no exception should be raised.
        Log.e(TAG_MAIN, "A malformed URL was provided for a remote certificate location");

}
```

For the rest the developer may proceed as shown in *Using SecureDexClassLoader to load dynamic code securely* and this procedure grants to succeed in the loading process for any of the two classes independently on the order in which they are attempted to be loaded.

Note: By design `SecureDexClassLoader` assumes that **each package name** is intrinsically related to a **single container**, while it is not necessary true the opposite. This means that attempting to *load a class*, whose **package name** is associated with **more than one container** provided in *dexPath* (i.e. each one of the two containers contains at least one class with the same package name), will generate an **unpredictable behavior** since `SecureDexClassLoader` will associate that package name with just one of the two containers.

So it is a **developer responsibility** to check the containers in order to avoid the occurrence of this rare but undesirable

situation.

4.3 Reverse package name to obtain remote certificate URL

Grab'n Run provides as an extra feature the possibility to **reconstruct the remote URL location of the certificate by reversing the package name** provided into the associative map. To enable this feature simply add an entry to the associative map where the **key** is the **desired package name to reverse** and the **value** is `null`. Here is a simple snippet of code to exemplify:

```
Map<String, URL> packageNamesToCertMap = new HashMap<String, URL>();

// Notice that a null entry won't raise a MalformedURLException..
packageNamesToCertMap.put("it.polimi.necst.mylibrary", null);
```

What is going on behind the curtains is that whenever GNR find an entry with *a valid package name associated to a null value*, it will **reverse the package name** with the following convention:

> The **first word** of the package name will be considered as the **top level domain (TLD)**, while the **second** one is going to be the **main domain**. Any **following word** of the package name will be used in the **same order** as they are listed to define the **file path** on the remote server and of course since a secure connection is needed for the certificate, **HTTPS protocol** will be enforced.

Let us translate this theory with some concrete examples:

- Package name `it` won't be reverted since it contains just a world (at least two are required for real world package name).
- Package name `it.polimi` will be reverted to the URL `https://polimi.it/certificate.pem`.
- Package name `it.polimi.necst.mylibrary` will be reverted to the URL `https://polimi.it/necst/mylibrary/certificate.pem`.

As you can see from the previous examples this naming convention assumes that the **final certificate** will be found in the *remote folder obtained by reverting the package name* and that the **certificate file** will have been **always renamed** `certificate.pem`.

4.4 Perform dynamic code loading concurrently

> **Warning:** Before approaching this paragraph, a good idea is having **first read** the Why should I use Grab'n Run? section of this documentation and in particular the last part on performance-related topics.

By default when a new `SecureDexClassLoader` object is instantiated, it will immediately **validate** all of its **containers concurrently** (**Eager signature verification strategy**). By the way sometimes when a large number of containers are assigned to a single `SecureDexClassLoader` object, it may just be *more convenient to evaluate each container separately just before loading classes from it*. So in such a scenario a **lazy signature verification strategy** would be advisable.

An even better *performance concern strategy* is loading target classes in a **concurrent** way on different threads. This is perfectly fine with *Grab'n Run* since the library is **thread-safe**.

As an example let us consider the case in which we want to *concurrently load some classes with a lazy strategy* from a `SecureDexClassLoader` instance with many containers associated to it. A possible code implementation which also makes use of Executors , FixedThreadPool and Future classes is the following:

```
// Make the assumption that packageNamesToCertMap has been already initialized;
// moreover longListOfDexPath is the String with all the containers path listed and
// separated by :
SecureLoaderFactory mSecureLoaderFactory = new SecureLoaderFactory(this);
// Initialize a SecureDexClassLoader instance in LAZY mode.
SecureDexClassLoader mSecureDexClassLoader =
        mSecureLoaderFactory.createDexClassLoader(      longListOfDexPath,
                                                        null,
                                                        getClass().getClassLoader(),
                                                        packageNamesToCertMap,
                                                        true);

// Suppose these classes belongs only to three different containers;
// while longListOfDexPath points to ten containers..
String[] classesToLoad = new String[] { "com.example.classA",
                                        "it.polimi.classB",
                                        "de.application.classC",
                                        "com.example.classD",
                                        "it.polimi.classE"};

// Suppose to store the loaded classes here..
Set<Class<?>> loadedClassesSet = Collections.synchronizedSet(new HashSet<Class<?>>());

// Initialize the thread pool executor with number of thread
// equals to the number of classes to load..
ExecutorService threadLoadClassPool = Executors.newFixedThreadPool(classesToLoad.size());
List<Future<?>> futureTaskList = new ArrayList<Future<?>>();

Iterator<String> classesToLoadIterator = classesToLoad.iterator();

while (classesToLoadIterator.hasNext()) {

        String classNameToLoad = classesToLoadIterator.next();

        // Submit a new class load thread on a container and store
        // a reference in the future objects list.
        Future<?> futureTask =
                threadLoadClassPool.submit(new classLoadingTask(mSecureDexClassLoader,
                                                                classNameToLoad,
                                                                loadedClassesSet));
        futureTaskList.add(futureTask);
}

// Stop accepting new tasks for the current threadLoadClassPool
threadLoadClassPool.shutdown();

for (Future<?> futureTask : futureTaskList) {

        try {

                // Wait till the current task for class loading is finished..
                futureTask.get();

        } catch (InterruptedException | ExecutionException e) {

                // Issue while executing the verification on a thread
                e.printStackTrace()
        }
```

```
}

try {

        // Join all the threads here.. Use a timeout eventually..
        threadLoadClassPool.awaitTermination(   KEEP_ALIVE_NUMBER_OF_TIME_UNITS,
                                                KEEP_ALIVE_TIME_UNIT);
} catch (InterruptedException e) {

        // One or more of the threads objects were still busy..
        // And this should not happen..
        e.printStackTrace()
}
```

And finally here it is the `classLoadingTask`, an implementation of the Runnable interface, which is responsible for **dynamically loading** a single class with the previously created `SecureDexClassLoader` instance. Here is the class implementation:

```
class classLoadingTask implements Runnable {

        // The shared instance of SecureDexClassLoader for concurrent load ops.
        private SecureDexClassLoader mSecureDexClassLoader;
        // The name of the class to load.
        private String classNameToLoad;
        // Concurrent set of class objects that were successfully loaded.
        private Set<String> successLoadedClassesSet;

        public classLoadingTask(        SecureDexClassLoader mSecureDexClassLoader,
                                        String classNameToLoad,
                                        Set<String> successLoadedClassesSet) {

                // Simply copy all the incoming parameters..
                this.mSecureDexClassLoader = mSecureDexClassLoader;
                this.classNameToLoad = classNameToLoad;
                this.successLoadedClassesSet = successLoadedClassesSet;
        }

        @Override
        public void run() {

                // Set current thread priority to DEFAULT.
                android.os.Process.setThreadPriority(android.os.Process.THREAD_PRIORITY_DEFAULT);

                try {

                        // Load operation is invoked..
                        Class<?> loadedClass = mSecureDexClassLoader.loadClass(classNameToLoad);

                        // Check whether the loading operation succeeds
                        if (loadedClass != null) {

                                // Class loading was successful and performed in a safe way.
                                // Add this class to the concurrent set
                                successLoadedClassesSet.add(loadedClass);
                        }

                } catch (ClassNotFoundException e) {
                        // This exception will be raised when the container of the
```

```
                // target class is genuine but this class file is missing..
                e.printStackTrace();
            } catch (InstantiationException e) {
                e.printStackTrace();
            } catch (IllegalAccessException e) {
                e.printStackTrace();
            }
        }

}
```

The interesting **advantage** of this *concurrent evaluation* is that **only the first loaded class** belonging to each separate container will perform the **signature verification** process when the `loadClass()` method is invoked, while all the other loaded classes from the same container will benefit from the cached result of this verification and so their evaluation will be way faster (comparable to the `loadClass()` time execution of `DexClassLoader`).

Note: Using this **concurrent lazy approach** is a good way to *lower the performance overhead* that may be introduced by *Grab'n Run* and *keep your application always responsive*. Another slight shrewdness that you may consider when you are in need to *load many classes from containers that have to be downloaded* is considering to show a ProgressDialog or a similar object to *make the user aware that your application is performing some tasks that require him/her to wait* and at the same time prevent the user from clicking everywhere or terminating your application since it sometimes may seem not fully responsive.

4.5 On library developer side: how to prepare a valid library container compatible with GNR

For once in this tutorial the **focus is now moved** from the *application developer*, who wants to load classes from an external library, **to the library developer**, who wrote a library and wants to make it available to the application developers.

What we are going to discuss about in this section is **how a library developer should prepare his/her library** in order to have it **compatible with GNR** system and more in general with **dynamic code loading**. A hint in this sense is provided by DexClassLoader documentation, which states clearly that this class, and so also `SecureDexClassLoader` does, *"loads classes from .jar and .apk files containing a classes.dex entry."*.

Note: The procedure outlined below must be performed entirely in case that you want to **export a library** into a *jar* container. The *typical use case* for such a situation is whenever you want to *export a library* which was *initially thought to work just for regular Java applications* but that now you would *also like to execute into an Android application*.

On the other hand, if you decide to **export an Android application** as a source for dynamic class loading, part of the upcoming procedure won't be necessary anymore. This happens because:

1. When an *apk* container is generated, `dx` tool is automatically invoked. This means that by considering a valid *apk* container as a source for classes to load, the `classes.dex` entry will be already present and so you won't need to manually execute step *1* and step *2* of the following guide.
2. Since Android requires an *apk* container to be signed to allow execution, you can decide, whenever you are ready to **export your application as a library**, to right click on the project and choose `Android Tools -> Export Signed Application Package...`. By completing the wizard procedure, you are going to export a signed version of the final *apk* container and this basically covers the first *4* steps of the following guide.

So let us assume that you, as a library developer, want to export your project called “MyLibrary” into a *jar* archive compatible with `SecureDexClassLoader`. The following steps should be performed:

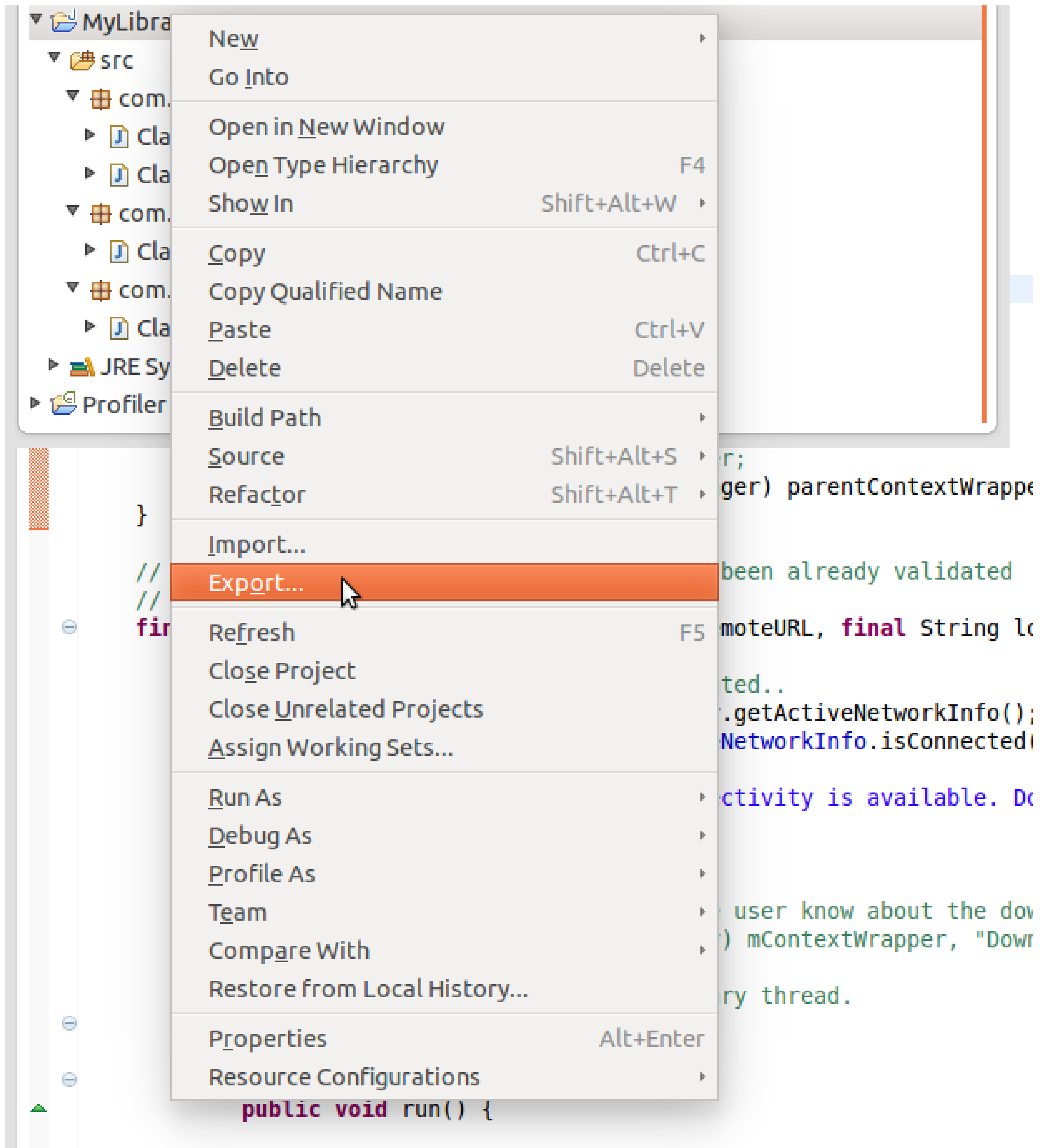
4.5.1 1. Export the project “MyLibrary” into a jar archive.

If your project was developed using **Android Studio**, you can easily obtain a copy of your *jar* library by opening a terminal and pointing it to the main folder of your project and then by invoking a series of tasks through the `./gradlew` script as shown here:

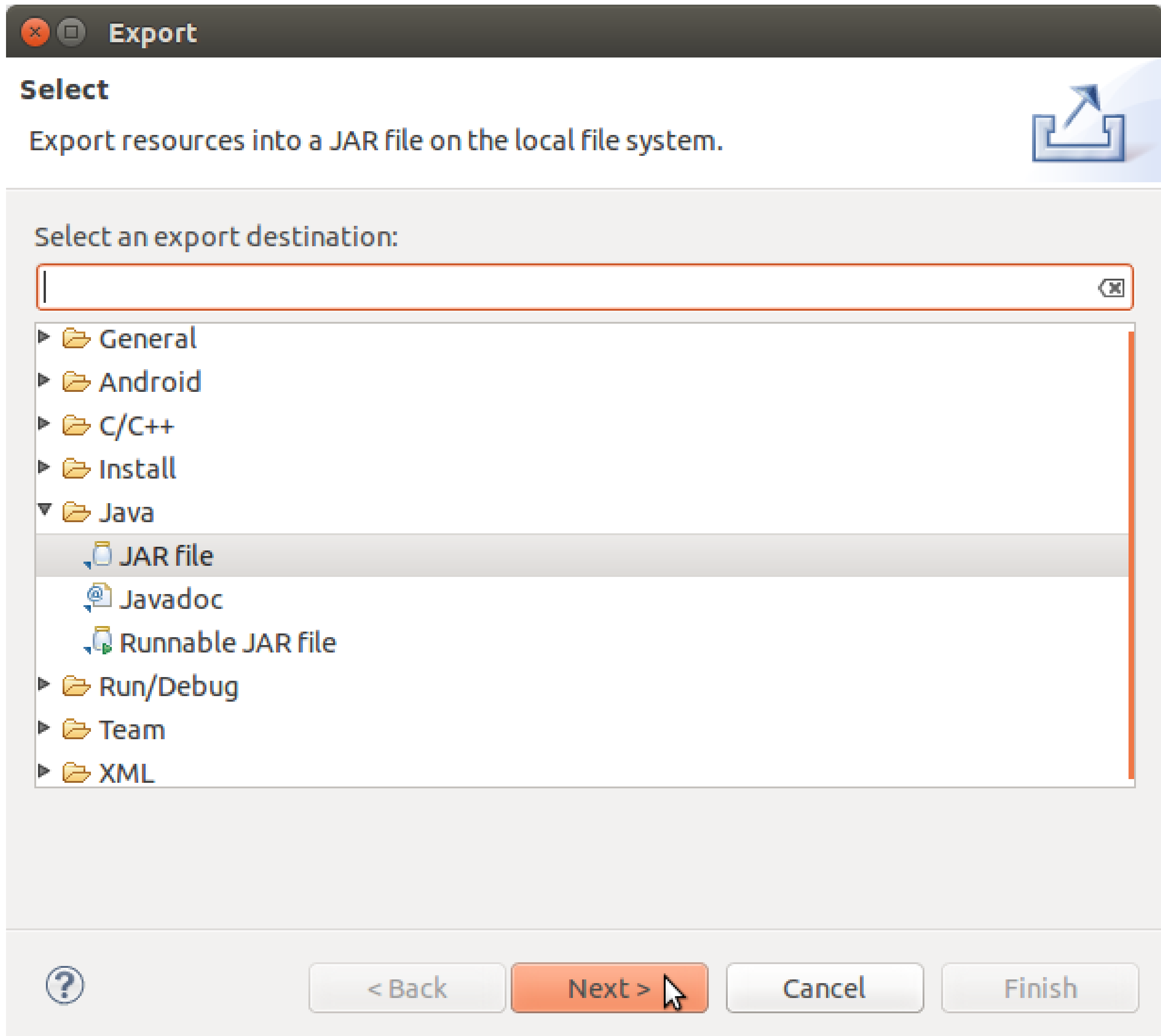
```
$ cd <absolute_path_to_your_jar_lib_project>
$ ./gradlew clean build assembleRelease
```

If the build process goes smoothly, you should now be able to find a file presumably called *“MyLibrary-release.jar”* located under one of your project `build/outputs` folder.

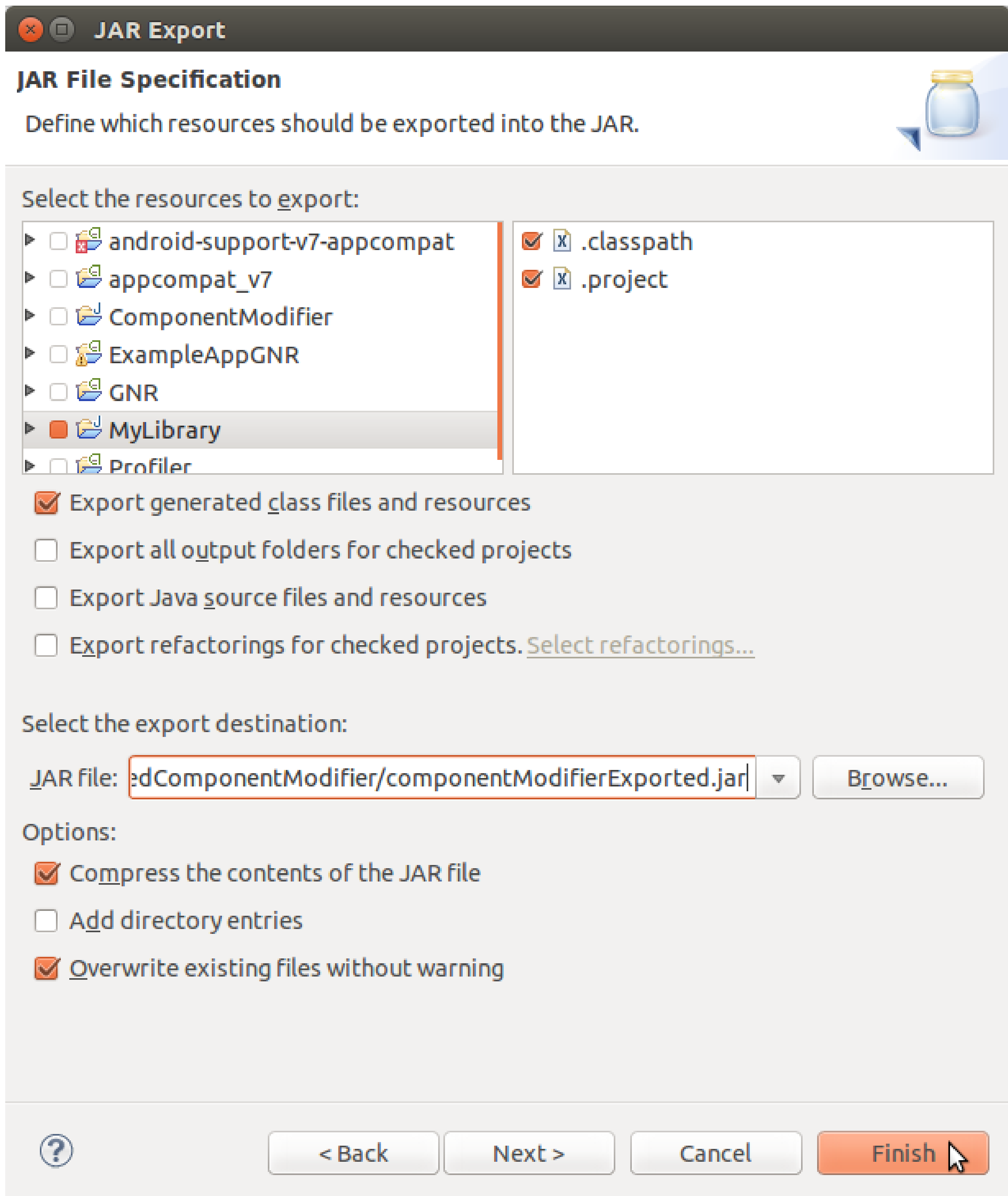
On the other hand if you are relying on the **ADT (Android Development Tool)**, right-click on the project *“MyLibrary”* and select “Export...”.



Then choose the option “Jar File” and click “Next...”.



Finally choose the location of the exported *jar* archive by clicking on the “Browse...” button and then “Finish”.



Independently from which of the two methods you implied, you should have now successfully exported your project into a *jar* container!

4.5.2 2. Translate Java Byte Code (.class) into Dalvik Byte Code (classes.dex).

After having exported your project into a *jar* container you now have code that can run on a **Java Virtual Machine (JVM)** in the form of class file with the extensions `.class`. Nevertheless in order to have your **code running** with `SecureDexClassLoader` **on an Android phone** it is necessary to **translate** the class files from Java Bytecode to **Dalvik Bytecode**. This task can be accomplished easily thanks to the `dx` tool, present in the Android SDK folder.

Note: Notice that **Dalvik Bytecode** is also compatible with the new Android Runtime (ART) system. This means that, except for narrow cases, you won't generally need to worry since your library code should execute fine on both the *Dalvik Virtual Machine (DVM)* and the *Android Runtime (ART)*. As related to this guide and more in general to *Grab'n Run, choosing one runtime system in stead of the other should not be an issue at all.*

So by assuming that you have just exported the project into a file called *myLibrary.jar* in a terminal type the following commands:

```
$ cd <path_to_exported_jar>
$ /<path_to_sdk>/build-tools/<last_stable_sdk_version>/dx --dex --output=myLibrary-dex.jar myLibrary
```

The result is an output *jar* container called *myLibrary-dex.jar*. You can easily spot that no `.class` file is stored in this container and in stead a file called `classes.dex` was added. This is the direct **result of the translation** mentioned before.

4.5.3 3. Generate a keypair and export the developer certificate

If this is the first time that you sign a container you will need to **generate a key pair** with `keytool` and then **export a certificate** containing the newly created public key. Otherwise if you *already have a key pair and the associated certificate, simply skip this section* and continue reading from the next one.

In order to **generate a keystore and a key pair** type in the following command line in a terminal:

```
$ keytool -genkey -v -keystore my-release-key.keystore -alias alias_name -keyalg RSA -keysize 2048 -v
```

This line prompts you for passwords for the keystore and private key, and to provide the Distinguished Name fields for your key. It then generates the keystore as a file called `my-release-key.keystore`. The keystore will contain a single key, valid for 10000 days. The **alias** is a name that you choose to **identify keys** inside the keystore. In this case this private key will be identified as `alias_name`.

If the previous step succeeded, now it is time to **export your developer certificate** that will be used by *application developers to verify your library code before dynamically loading it*. This can be accomplished again thanks to a `keytool` feature:

```
$ keytool -exportcert -keystore my-release-key.keystore -alias alias_name -file certificate.pem
```

This command will export the certificate embedding the public key associated to the private key whose alias is `alias_name`. This certificate will be stored in the file `certificate.pem`.

Even if the previous commands are all that you will need here, if you desire to deepen your knowledge on *keystore, keys and signing Android applications* visit these reference links:

- https://www.digitalocean.com/community/tutorials/java-keytool-essentials-working-with-java-keystores
- http://developer.android.com/tools/publishing/app-signing.html#signing-manually

4.5.4 4. Sign the library with the developer private key.

Now it is time to **sign** the *jar* library with the **library developer private key** to enable the possibility to verify it.

Assuming that you have generated a private key whose alias is `alias_name` and stored it in a keystore whose name is `my-release-key.keystore` in order to sign the *jar* container manually type in this line in your terminal:

```
$ jarsigner -verbose -sigalg SHA1withRSA -digestalg SHA1 -keystore my-release-key.keystore myLibrary-
```

You can then verify that the jar container is actually signed by typing:

```
$ jarsigner -verify -verbose -certs myLibrary-dex.jar
```

Note: When you verify the signature of the final container, you will receive a **warning message** like the following *"This jar contains entries whose certificate chain is not validated"*. This is absolutely normal since a **self-signed certificate** was used for the **verification process** and this is acceptable in Android as long as you are absolutely *sure that the certificate used for the verification is actually the library developer one*. In *Grab'n Run* the **chain of trust** is replaced by assuming that the certificate is stored on a domain which is directly controlled by the library developer and can only be retrieved via **HTTPS protocol**.

4.5.5 5. Make the library and the certificate publicly available.

The last step is **making public the signed version of the jar container**, obtained after the previous step, and the **exported certificate** embedding the library developer public key (*as explained in 3. Generate a keypair and export the developer certificate*).

While you can *store the library container basically everywhere on the web* (application developers can retrieve your library via both HTTP or HTTPS protocol), it is **crucial and fundamental** for the whole security model to handle that you **publish your library developer certificate on a secure trustworthy remote location which can be accessed only via HTTPS protocol**.

If you have successfully followed up all the previous steps, you have now correctly **published your library** and application developers will be able to **run your code securely** by using `SecureDexClassLoader`.

4.6 Let GNR automatically handle library updates silently

In the end of this section **silent updating**, a *powerful feature* of **dynamic code loading**, is presented and easily and securely implemented with the use of *Grab'n Run*. Performing silent updates is a convenient techniques which can be used to **keep always updated third-party libraries or frameworks** by *decoupling the update process of the main application from those ones of the non-standalone libraries*. The **advantage** of such an approach is clearly the possibility to have always the **latest features and security workaround on third-party libraries** without continuously bothering the user on updating the application.

Dynamic code loading in this sense can be really effective in such a scenario since the latest version of the code can be retrieved from a remote URL just at runtime and then immediately executed.

Let us now set up a possible use case for this technique and see how to implement it with Grab'n Run from both the library developer and the application developer side: imagine that an application developer wants to dynamically load the latest version of the already seen class `com.example.ClassA` stored in "myLibrary-dex.jar", a remote library.

From the point of view of the **library developer** a couple of prerequisite steps must be performed:

- The developer must prepare correctly a **signed version** of his/her library. For a complete walk-through on this task see the previous section *On library developer side: how to prepare a valid library container compatible with GNR*.

- Once that the last version of the library container is correctly prepared and signed, the **developer must publish** on a domain that (s)he controls a **redirect link** (i.e. `http://mylibrary.it/downloads/mobile/latest`) which *points to the remote location where the library container is actually stored* (i.e. `http://mylibrary.it/downloads/mobile/myLibrary-dex-1-8.jar`).
- The developer must also set up a **secure link using HTTPS protocol**, which *points to the remote location of the certificate* associated to the private key used to sign the last version of the library (i.e. `https://myLibrary.com/developerCert.pem`).
- Every time that a **new version of the same library is ready** (i.e. version 1.9 of myLibrary is now available), the library developer will have to prepare the container in the usual way and sign it with the **SAME** private key associated to `developerCert.pem` and finally **update the redirect link** to *point to the location of the latest version of the container* (i.e. set up `http://mylibrary.it/downloads/mobile/latest` to redirect to `http://mylibrary.it/downloads/mobile/myLibrary-dex-1-9.jar`).

Warning: While *Grab'n Run* **supports redirect links for the container remote location**, this kind of link is arbitrarily not accepted for remote certificates!!! This is a **security-oriented choice** since redirect links may jump from an HTTPS link to an HTTP one making the whole system insecure in case that the attacker performs a **Man-In-The-Middle-Attack** and substitute the proper certificate for the verification with a different one generated by himself. That is the reason why **redirect links for remote certificates will not be followed** by Grab'n Run and so no certificate file will be found for the container signature verification.

On the other hand the **application developer**, who wants to make use of the classes provided by `myLibrary` can easily accomplish this by setting up a `SecureDexClassLoader` where the *location pointing to the remote container* is the **redirect link** provided by the library developer and the **certificate** used for the verification is the *one stored at the secure URL on the library developer domain*. Here is a snippet of code that summarizes this operational description:

```
ClassA classAInstance = null;
// The latest version of the library container is always found thanks to the redirect link
jarContainerRemotePath = "http://mylibrary.it/downloads/mobile/latest";

try {
        Map<String, URL> packageNamesToCertMap = new HashMap<String, URL>();

        // The package "com.example" is always signed by the library developer with
        // the same private key and so it can always be verified with the same
        // remote certificate.
        packageNamesToCertMap.put(      "com.example",
                                        new URL("https://myLibrary.com/developerCert.pem"));

        // The second parameter used here specifies how many days are counted before
        // a cached copy of the remote library container is considered rotten
        // and automatically discarded.
        // Default value is 5 days, here the value is lowered to 3..
        SecureLoaderFactory mSecureLoaderFactory = new SecureLoaderFactory(this, 3);
        SecureDexClassLoader mSecureDexClassLoader =
                mSecureLoaderFactory.createDexClassLoader(      jarContainerRemotePath,
                                                                null,
                                                                packageNamesToCertMap,
                                                                getClass().getClassLoader());

        Class<?> loadedClass = mSecureDexClassLoader.loadClass("com.example.ClassA");
```

```
        // Check whether the signature verification process succeeds
        if (loadedClass != null) {

                // Class loading was successful and performed in a safe way.
                // The last version of ClassA has been successfully retrieved!
                classAInstance = (ClassA) loadedClass.newInstance();

                // Do something with the loaded object classAInstance
                // i.e. classAInstance.doSomething();
        }

} catch (ClassNotFoundException e) {
        // This exception will be raised when the container of the target class
        // is genuine but this class file is missing..
        e.printStackTrace();
} catch (InstantiationException e) {
        e.printStackTrace();
} catch (IllegalAccessException e) {
        e.printStackTrace();
} catch (MalformedURLException e) {
        // The previous URL used for the packageNamesToCertMap entry was a malformed one.
        Log.e("Error", "A malformed URL was provided for a remote certificate location");
}
```

CHAPTER 5

Repackaging tool

In this section you will learn how to configure and use the repackaging tool to have your applications rewritten automatically to use the Grab'n Run library. The repackaging tool allows you to:

- Rewrite your application to use Grab'n Run secure API instead of the standard Android ones. This process is automatic and requires only a couple of settings from your side.
- Granular control on the security policy that you want your application to follow at run time when performing dynamic code loading (e.g., validate all the dynamically loaded code against one certificate, validate each container against a different certificate, decide whether to provide a default trusted certificate for all the not specified entries).

This tool relies on Androguard to decoded, decompile, and rebuild the application provided as input.

A prerequisite for using the tool is that your local machine must be able to have Internet connectivity since it will be necessary for the tool to download the source containers declared as potential sources for dynamic code loading unless they result directly accessible on the file-system of your machine.

Another prerequisite is that you have already successfully installed apktool on your local machine.

5.1 Use

1. Open a terminal and move to the repackaging tool script *repackPOC* directory and then install the required dependencies:

```
$ cd <absolute_path_to_GNR_folder>/repackPOC
$ pip install -r requirements.txt
```

2. Run the script:

```
$ python repackagingTool.py
```

3. For a list of the optional arguments:

```
$ python repackagingTool.py -h
```

5.2 Configuration

Before having the repackaging tool executed, you will have to go through a couple of windows to configure how you want your application to be patched. In particular:

1. You have to select which application, in the form of an APK container, you want to patch.
2. You have to provide the JAR or APK containers used as sources for dynamic code loading.
3. You need to specify a security policy that will be used to evaluate whether the containers used as sources are genuine at runtime.

You can customize these settings easily thanks to a simple UI presented whenever the script is started.

5.2.1 Step 1: Select the application to patch

Select the application you want to patch to use the secure Grab'n Run APIs. Notice that this application must be a regular APK file, which is stored locally on your machine.

It is not required for this container to be aligned, or signed, but notice that you will *have to* sign, and then align the final application generated by the script in order to have it installed on the user's devices.

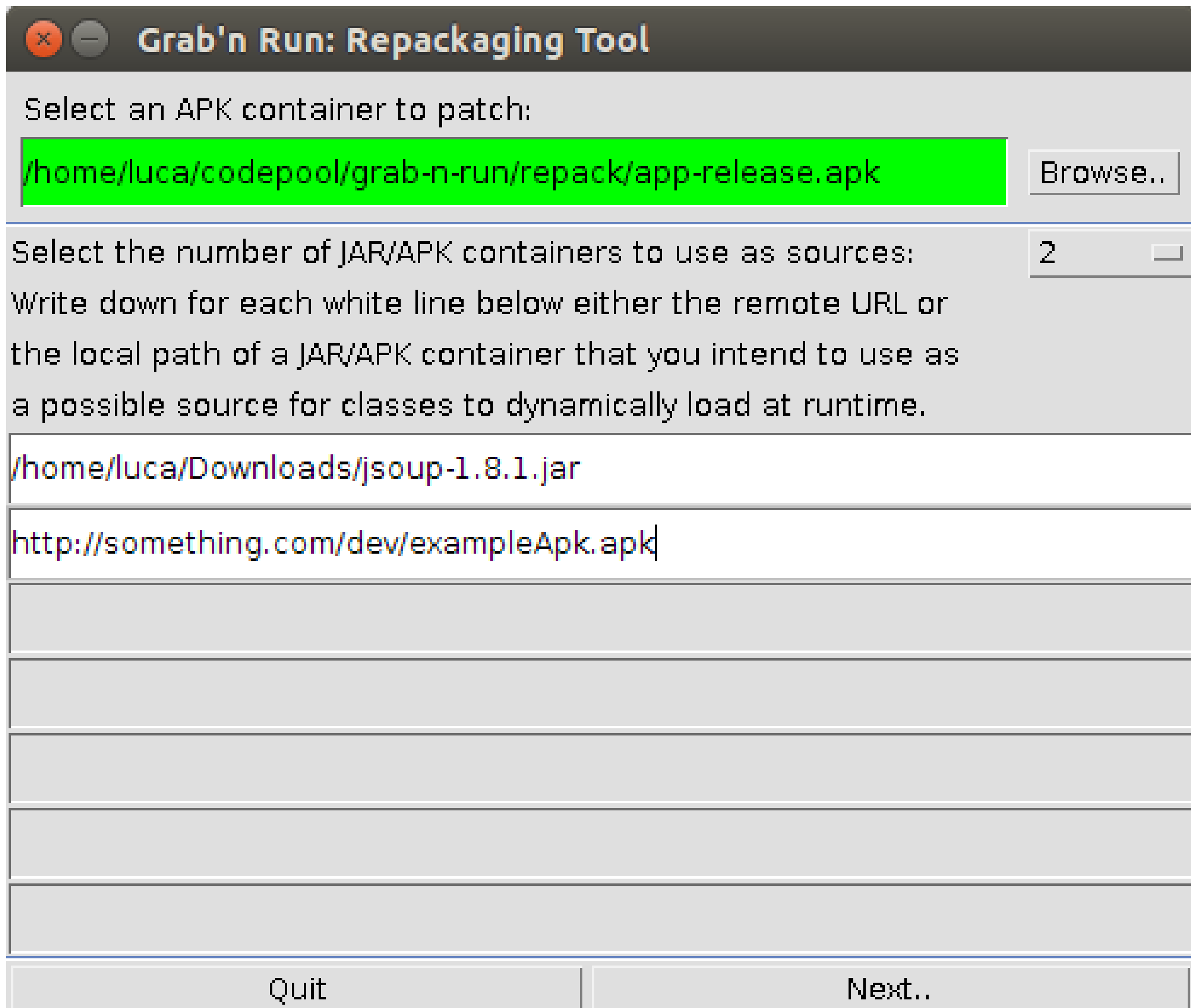
If you are not familiar on how to *sign*, and *align* an APK, you may want to check the third bullet point of the "Quick example of use" section in the `README` file of the Grab'n Run repository on GitHub.

5.2.2 Step 2: Select the containers sources for DCL

Next step is listing all the JAR, or APK containers used by your application as sources for dynamic code loading. Each container can be either stored locally (in this case indicate the absolute path on the file system), or remotely on an endpoint (in this case simply type the URL pointing to the file, and notice that both HTTP and HTTPS protocol are supported).

Warning: In case of a JAR contianer, verify that you have already translated its content into Dalvik bytecode. Containers that have been gone through this process present a specific entry named `classes.dex`. If you provide a container without such an entry, the repackaging tool will simply ignore it.

The screenshot below shows the first screen of the UI with an example of a feasible configuration for Step 1 and Step 2.

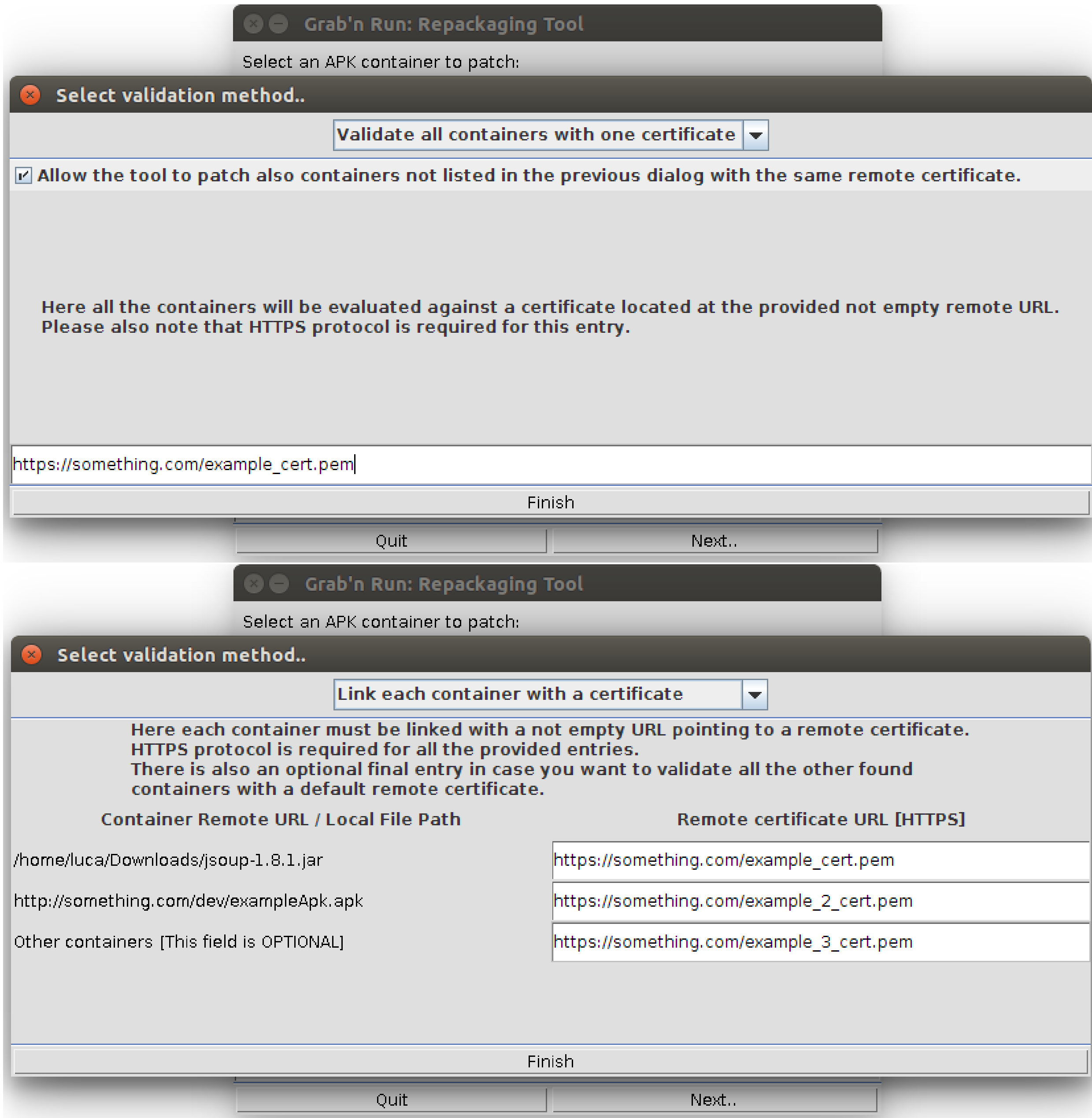


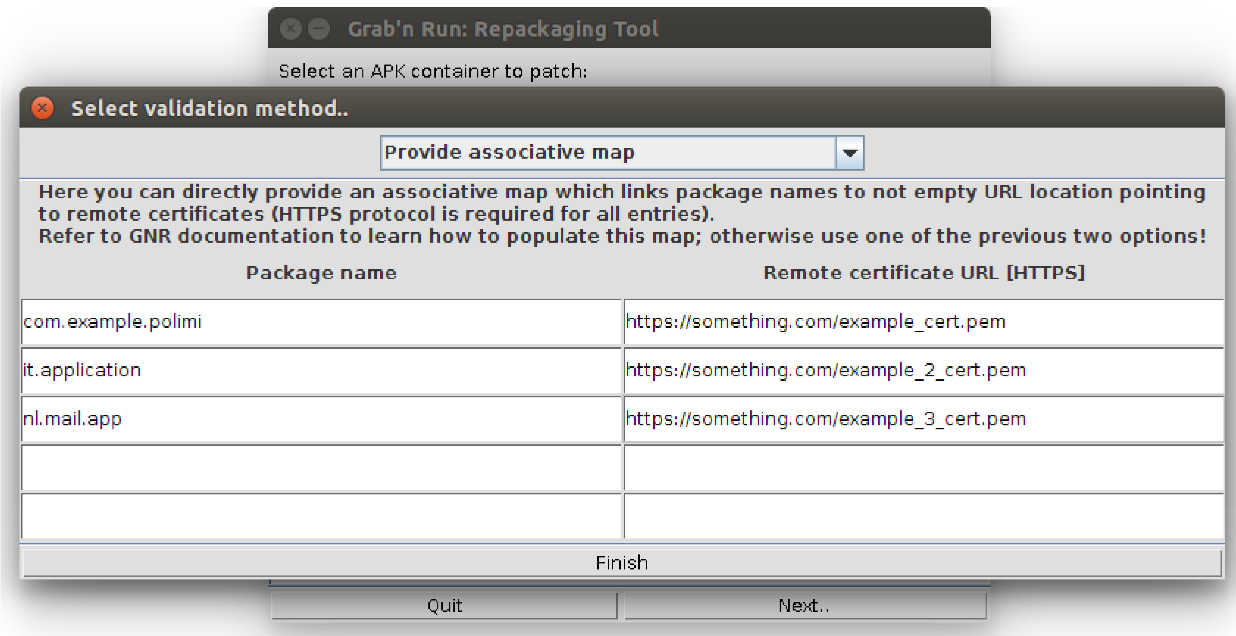
5.2.3 Step 3: Configure the security policy for the source containers

Once you confirm the settings in the first window, you will be prompted with a new screen to decide against which certificate each source container should be verified by Grab'n Run. At the moment we support three policies:

1. Evaluate all the containers against one certificate.
2. Evaluate each container against a different certificate.
3. Provide a validation mapping, which has on the left side the prefix of the package name of the involved classes, and on the right side the corresponding certificate to use for the evaluation.

The three screenshots below show an example of a valid configuration for each of the three policy. Switching from one to another is as simple as picking a different value in the drop-down menu box on top of the UI.


Grab'n Run: Repackaging Tool
Select an APK container to patch:
Select validation method..
Validate all containers with one certificate
Allow the tool to patch also containers not listed in the previous dialog with the same remote certificate.
Here all the containers will be evaluated against a certificate located at the provided not empty remote URL.
Please also note that HTTPS protocol is required for this entry.
https://something.com/example_cert.pem
Finish
Quit
Next..
Grab'n Run: Repackaging Tool
Select an APK container to patch:
Select validation method..
Link each container with a certificate
Here each container must be linked with a not empty URL pointing to a remote certificate.
HTTPS protocol is required for all the provided entries.
There is also an optional final entry in case you want to validate all the other found
containers with a default remote certificate.
Container Remote URL / Local File Path
Remote certificate URL [HTTPS]
/home/luca/Downloads/jsoup-1.8.1.jar
https://something.com/example_cert.pem
http://something.com/dev/exampleApk.apk
https://something.com/example_2_cert.pem
Other containers [This field is OPTIONAL]
https://something.com/example_3_cert.pem
Finish
Quit
Next..



Some further considerations for this step:

- You must provide each certificate through a remote reference to a secure endpoint containing such resource. The use of HTTPS protocol for the endpoint is required, and, if this is not the case, the repackaging tool will enforce it.
- For the first two policies you can select as source containers either archives stored locally on your machine (use the absolute path of these resources on the file-system), or a remote endpoint containing such container (use its URL). In the latter case the repackaging tool will take care, when started, to retrieve these containers for further analysis. For these two policies, you can optionally also provide a default certificate that will be used for evaluating any other source container, which was not listed in the previous step but is used by your application at run time.
- For the third policy the most restrictive package name prefix will be applied during the evaluation: This means that if your application wants to load code for a class, whose full name is `com.example.myapp.MyClass`, and you have two entries in your mapping, one for `com.example` and the other for `com.example.myapp`, the certificate associated to the latter one will be used since this entry has a longer prefix matching the name of the class to load. You should consider to use the last policy only if you are really aware of how the system works (see Quick start and tutorial, and Complementary topics for further details on the mapping process); in general, using one of the two first policies is already enough for most of the use cases, and it avoids pain from your side since the tool handles automatically the process of generating a validation mapping from your settings.

5.2.4 Step 4: Gotta patch them all!

Once you enter your settings and you press the "*Finish*" button, the repackaging tool will start its execution. If no error is raised, the patched APK will be available in the main folder where you launched the script from terminal (take extra care of having the original application in a different folder unless you want it to be overwritten by the patched version).

I hope you will find this tool useful and I am eager to hear your feedbacks :)

CHAPTER 6

Indices and tables

- genindex
- modindex
- search